AFRL-HE-WP-TR-2006-0153

Specification for Visual Requirements of Work-Centered Software Systems

James R. Knapp

Wright State University
3640 Colonel Glenn Highway
Dayton OH 45435-0001

October 2006

Interim Report for November 2004 to October 2006

Air Force Research Laboratory
Human Effectiveness Directorate
Warfighter Interface Division
Cognitive Systems Branch
WPAFB OH 45433-7604

NOTICE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory, Human Effectiveness Directorate, Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-HE-WP-TR-2006-0153 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR

//SIGNED//

DANIEL G. GODDARD
Chief, Warfighter Interface Division
Air Force Research Laboratory

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
October 2006	Thesis	November 2004 - October 2006

4. TITLE AND SUBTITLE
Specification for Visual Requirements of Work-Centered Software Systems

5a. CONTRACT NUMBER
F33601-03-F-0064

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
63231F

6. AUTHOR(S)
James R. Knapp

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
28300310

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Wright State University
3640 Colonel Glenn Highway
Dayton OH 45435-0001

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Materiel Command
Air Force Research Laboratory
Human Effectiveness Directorate
Warfighter Interface Division
Cognitive Systems Branch
Wright-Patterson AFB OH 45433-7604

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/HECS

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-HE-WP-TR-2006-0153

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distributed is unlimited.
Cleared by AFRL/PA as AFRL/WS-06-2458 on 10 October 2006.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Work-centered software systems function as inherent work-aiding systems. Based on the design concept for a work-centered support system (WCSS), these software systems support user tasks and goals through both direct and indirect aiding methods within the interface client. In order to ensure the coherent development and delivery of work-centered software products, WCSS visual interface requirements must be specified in order to capture the cognitive and work-aiding aspects of the user interface design. Without the ability to specify such original requirements, the probability of creating an accurate and effective work-centered software system is significantly reduced. A new visual requirements specification language based on the User Interface Markup Language (UIML) is proposed as an effective solution to bridging this gap between cognitive systems engineering and software engineering. In this paper, a new visual requirements specification language that can capture and describe work-centered visual requirements within a semi-formal syntax is introduced and explained. The proposed language is also shown to be easily integrated into a UML object model via the use of UML's extensibility features. Such a specification language for visual requirements could be employed by cognitive engineers and design teams to help convey requirements in a comprehensible format that is suitable for a software engineer. Such a solution provides coherency in the software modeling process of developing work-centered software systems and contributes towards the specification of unique visual software requirements.

15. SUBJECT TERMS Work-Centered Support Systems, Work-Centered Software Systems, User Interface Markup Language

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Vincent A. Schmidt
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR	116	19b. TELEPHONE NUMBER (include area code)

THIS PAGE LEFT INTENTIONALLY BLANK

SPECIFICATION FOR VISUAL REQUIREMENTS OF WORK-CENTERED SOFTWARE SYSTEMS

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Science

By

JAMES ROBERT KNAPP B.S.,

Wright State University, 2005

2006

Wright State University

WRIGHT STATE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

October 18, 2006

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY James Robert Knapp ENTITLED Specification for Visual Requirements of Work-centered Software Systems BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science.

_____

Soon Chung, Ph.D.
Thesis Director

_____

Forouzan Golshani, Ph.D.
Department Chair

Committee on
Final Examination

_____

Soon Chung, Ph.D.

_____

Thomas Hartrum, Ph.D.

_____

Vincent Schmidt, Ph.D.

_____

Joseph F. Thomas, Jr.
Dean, School of Graduate Studies

ABSTRACT

Knapp, James Robert. M.S., Department of Computer Science and Engineering,
Wright State University, 2006.
Specification for Visual Requirements of Work-Centered Software Systems.

Work-centered software systems function as inherent work-aiding systems. Based on the design concept for a work-centered support system (WCSS), these software systems support user tasks and goals through both direct and indirect aiding methods within the interface client. In order to ensure the coherent development and delivery of work-centered software products, WCSS visual interface requirements must be specified in order to capture the cognitive and work-aiding aspects of the user interface design. Without the ability to specify such original requirements, the probability of creating an accurate and effective work-centered software system is significantly reduced. A new visual requirements specification language based on the User Interface Markup Language (UIML) is proposed as an effective solution to bridging this gap between cognitive systems engineering and software engineering. In this paper, a new visual requirements specification language that can capture and describe work-centered visual requirements within a semi-formal syntax is introduced and explained. The proposed language is also shown to be easily integrated into a UML object model via the use of UML's extensibility features. Such a specification language for visual requirements could be employed by cognitive engineers and design teams to help convey requirements in a comprehensible format that is suitable for a software engineer. This solution provides coherency in the software modeling process of developing work-centered software systems and contributes towards the specification of unique visual software requirements.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

This research was supported in part by a research fellowship jointly provided by the Air Force Research Laboratory and the Dayton Area Graduate Studies Institute. This research was also supported in part by an appointment to the Research Participation Program at the Air Force Research Laboratory, Human Effectiveness Directorate, Bioscience and Protection, Wright Patterson AFB administered by the Oak Ridge Institute for Science and Education through an interagency agreement between the U.S. Department of Energy and AFRL/HEP.

This document is cleared for Public Release by AFRL/WS Public Affairs, AFRL-WS 06-2458.

THIS PAGE LEFT INTENTIONALLY BLANK

1. INTRODUCTION

Software engineering is an engineering discipline covering the lifecycle of software production from start to finish. It encompasses a large number of components to support this production including the use of: theories, methodologies, tools, languages, and management techniques. Each of these unique components is integrated throughout the software lifecycle in an effort to produce a robust software product. Software engineering adopts a systematic and organized approach as the most effective way to produce high-quality software [1]. In order to produce the software product, a set of activities and associated results are completed in what is known as a software process or software modeling process [1]. There are several major steps in any particular software process. These steps occur chronologically as progress is made towards a finished product. Every system requires a different and unique software process to best suit its individual needs. One of the first major milestones in a software process is the specification of the software to be developed. This entails the definition of the software's operation as well as constraints upon that operation. This crucial first step in the software modeling process leads directly into development, implementation, and other further steps. A good software process is one which reliably communicates information from one step to another, laying a foundation for coherence across the entire process.

The Unified Modeling Language (UML) has shown itself to be an effective method employed in the software modeling process. Since its initial standardization in

early 1997, the UML is the most widely used modeling approach for contemporary software engineering. The UML provides a versatile starting point for covering the various facets of the software development process. The premise for the creation of the UML was fundamentally one of communication [2]. Without a standard by which engineers could effectively communicate, the growth of the field of software development was considerably handicapped. The UML accomplished this standard in many ways including: establishing a common medium for communication across stakeholders and development team members, acting as a repository that documents incremental development decisions, and providing a mechanism in which to convey design specifications for final implementation.

The UML is composed of a graphical notation and corresponding meta-model [2]. The graphical notation is the general syntax used in the various model diagrams which the UML uses to display aspects of system behavior. The composition of all these diagrams gives an overall object model of the software to be developed. The UML notation is the visual portion of the language, while the meta-model, which defines the concepts of the language itself, provides the back-end framework. After capturing the various requirements of a system through elicitation and analysis, they can be mapped into the UML's library of diagrams. The collection of all the created diagrams known as the object model serves as a specification which forms a basis for implementation once fully conceived.

The UML possesses valuable assets in being able to continue to grow and extend its capabilities to meet new software development needs. In 2003, UML 2.0 became accepted as the new UML standard, including three brand new diagrams to aid in modeling behavior. Additions such as this recent upgrade give strength to the UML's versatility and show that it has potential for the long term.

The need for powerful, high-quality software in today's world is of crucial importance. The amount of data presented to users in new software continues to grow as software is developed to accomplish more complex work tasks. Along with total data, the level of computational complexity for the user is also rising rapidly. Software users must perceive, absorb, and make more complicated decisions within software than ever before. Government agencies such as the Department of Defense are migrating towards net-centric environments where data repositories can be fused together to form massive information hubs. Net-centric environments, therefore, are likely to further exacerbate this information overload problem. Progress is being made against this issue in proposed solutions such as the Joint Battlespace Infosphere (JBI), which allows operators to subscribe to data sources using data fusion tools to filter relevant information. However, this approach and others like it do not ensure that appropriate views of the filtered data are work supportive or give an initial work environment representation which can then be customized to the work being done [3]. In order for software to be successful in military, business, and other applications it must be functionally adept to accomplish its tasks as well as helpful and convenient to its end-users in completing those tasks.

In the early years of software engineering development, the focus of system design was primarily to create a product which was functionally operational. The creation of a piece of software which achieved a specific task was considered a successful and worthy investment. As time went on, the paradigm shifted as system developers realized that there must exist certain usability requirements in the development of the product in order to ensure that it can be understood and used by the operator. This led to what is referred to as user-oriented design, which is, in effect, design from beginning to end with the end-user in mind. However, as aforementioned

in the case of data fusion, giving the user more options and control does not necessarily help him to accomplish work. Developing software systems with a focus on work has been researched and defined by Eggleston et al. as a work-centered support system. According to their definition, a work-centered support system design approaches work representation in terms of how workers see and engage work [4]. This work representation effectively captures the work ontology, which is essential for building software around the work environment. Within this theory, software is developed under a work-oriented framework, allowing components such as the software interface client to be developed as a work support aid. The interface client takes the form of a customized graphical representation of the user's work environment, allowing the user to comprehend and employ the software most effectively. As a system is conceptualized, it is made to implicitly support the user in completing work. In order for software capabilities to be fully maximized towards performing work in the field of practice, they must be developed from a work-centered design methodology.

From a work-centered point of view, the UML shows inadequacy in its ability to model work-oriented behavior. Although a powerful tool, the UML has no specific modeling of user goals and intentions, showing it to be inept in expressing usage-oriented functionality [5]. The UML was not designed to be an all encompassing modeling tool, and displays an overall lack of support in the development of a work-centered object model. In general, the UML's methods are relatively informal, emphasizing usefulness rather than precision. The UML serves to highlight the important details and retain the most desired features in the development of a system. However, since the UML is the primary software engineering technique used by current system developers, this deficiency in expressing usage-oriented functionality

keeps many systems from being designed in an optimal work-centered manner. This gap that exists in the UML correlates to a more abstract need for a direct link between software engineering and cognitive systems engineering in this area. It is important to note that while the UML is not all sufficient it does retain the possibility for further extensions and enhancements, and even encourages such augmentation. Many experts seem to agree that any perceived gaps could be bridged by making alterations and additions to the already standard and robust UML [5].

In producing work-centered software, 2 major development gaps prevent projects from attaining successful product completion. These key hindrances are illustrated in Figure 1.1.

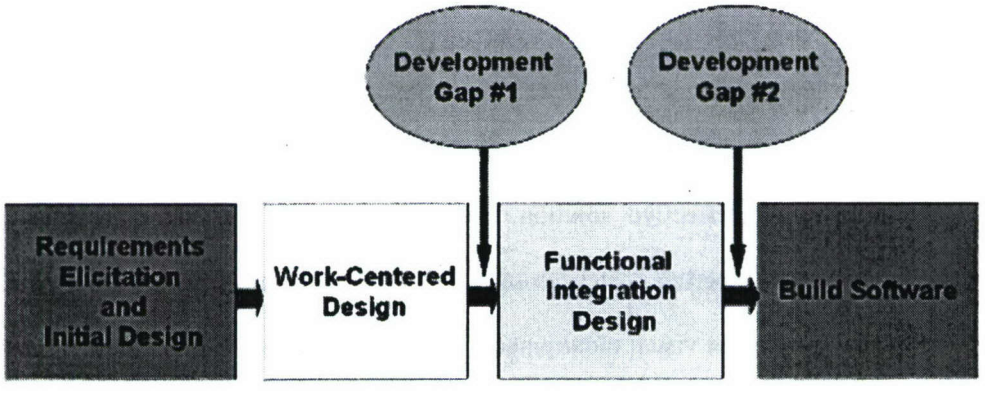


Figure 1.1 Work-centered Software Development Gaps

The first gap depicts the need for a method of capturing visual work-centered requirements so that they may be verified and accurately communicated to a developer after the work-centered design has been conceived. The second gap reveals the need to combine unique work-centered requirements specification with other standard specification such as the UML. These deficiencies deter the production of work-centered software and dramatically decrease project coherency. The result is a highly ad-hoc and chaotic software process leading to a high percentage of

miscommunication. These gaps in visual requirements specification must be met in order to enable a stable and repeatable work-centered software process.

Current software engineering processes and methods have difficulty expressing certain types of requirements throughout development. Many of these are what are known as non-functional or quality requirements. These types of requirements often involve factors closely associate with user work. The user interface or visual portion of the software product falls firmly into this category of being supportive to work tasks. The UML does not currently support this representational layer of the design. For work-supporting software to become a reality, the ability to specify work-centered requirements from a software engineering perspective must be made available. Otherwise, vital graphical interface requirements will be lost during implementation.

This paper proposes a visual requirements specification language as an intuitive and effective solution to bringing cognitive systems engineering and software engineering a step closer together. The language employs a semi-formal syntax to present visual requirements in a structured and unambiguous format. Using the visual requirements specification language, designers can formally capture work-centered visual requirements which are essential to successful work-aiding interface development. This document also elaborates on how the proposed language easily integrates into a UML object model via the use of augmentation capabilities. This document covers all the relevant aspects necessary to understanding and defining the new specification language. Chapter 2 provides a more in-depth background of the parties involved in the problem. It discusses work-centered support system concepts and current software engineering requirements collection techniques in further detail. Chapter 3 describes existing work-centered software products that have been or are

currently being developed by the Air Force Research Laboratory. Chapter 4 covers the details of the new visual requirements language. This includes its foundations, framework, syntax and semantics. Chapter 5 gives an example of what a practical use of the specification would look like using one of the software products described in chapter 3. Chapter 6 explains how the specification language integrates into a UML object model. It also covers how doing work-centered specification fits into a standard software process. Chapter 7 provides a review of the contributions of the proposed language. It also covers related topics and gives future direction for research. Two appendices are provided with additional materials regarding the contents of this paper. Appendix A shows a step-by-step method for employing the visual specification language to specify a work-centered user interface. Appendix B gives the full specification of the software product partially covered in chapter 5.

2. BACKGROUND

2.1 Work-centered Support Systems

A work-centered support system (WCSS) is a software system composed of a group of interacting elements focused on helping the user to accomplish work. As described by Eggleston et al., "a WCSS appears as a graphical user interface with embedded support tools in a work-centered organizational structure" [6]. WCSS is based upon the concept that the primary purpose of a software user interface is to function as a work aiding system. It combines representational aiding with intelligent automation within a single organizing framework [7]. WCSS is both a design technology and an interface client technology for the user interface layer of software application [6].

2.1.1 WCSS Interface Client Technology

As an interface client technology, WCSS dictates portions of control structure, object model, and user interface in a software product [6]. In normal practice, the form of the user interface is dominated by concerns over information object design, incorporation of good human factors, and meeting general style guidelines for human-computer interaction. However, little effort has been devoted to treating the interface as a support system in its own right [4]. Modern software systems contain many characteristics which inhibit the accomplishment of work tasks. Users often suffer from information overload, a condition where the user is inundated with information

used to complete work tasks. Military systems such as the Joint Battlespace Infosphere (JBI) support massive data fusion and selection, and therefore are highly susceptible to information overload. Although such architectures provide the user with all the knowledge necessary to make work decisions, there is no guarantee that the user will be aided by the format in which information is presented. On the contrary, it is more likely that the gross amount of data will burden the worker in his ability to perform. Another common detriment to software user interface technology is automation surprise. This issue concerns unexpected and confusing user interface behavior. The result of such activity diverts attention away from work tasks and leads to performance errors and costly time delays. The WCSS approach achieves effective support for these cognitive concerns by blending various aiding tools in a manner that is tailored to the characteristics of user work [6]. In this form, the speed and quality of task decision making is improved and the amount of cognitive burden placed on the user is minimized.

By highlighting and representing the key features of the work domain, the interface is made sensitive to the work context and able to support the range of work assigned to the user. This includes methods of both direct and indirect work support. Direct aiding is provided by a coordinated set of software agents that interact with the user and are clearly connected to or embedded in the work domain visualizations [8]. Indirect aiding is provided largely through the use of work domain visualizations and common work terms [8]. The main ingredients which constitute a work-centered support system are: a set of representational forms that themselves act simultaneously as work aids and GUI panels (for perceptual-based analysis and situational awareness), a set of different classes of software agents crafted and made available to

automatically perform work tasks under the guidance and control of the user, and a common work domain ontology to connect the various forms of aiding.

Representational forms present work tasks in domain terms, showing the problem state, environment constraints, and resources available for their completion [7]. These types of aids are context relevant, meaning they attempt to capture the work domain instead of simply being an activity-based model. When the work domain is used as a base point for providing support, the aid better accommodates the flexible and adaptive nature of user work [7]. The user is then capable to address complex work situations without suffering from complicated reasoning. Representational forms are supplied by the graphical and visual portions of the user interface. This includes the necessary components for the work domain to be represented within the software support tool.

Software agents handle the automation and data fusion portions of a WCSS. These agents provide a form of direct aiding although they may or may not always be visible to the user. Each agent can automatically perform work tasks with the user's permission. Agents give unique, individual aiding for functional elements of work such as data transformation and computation. Overall, the agent store functions to reduce the cognitive processing demands on the user.

In order to achieve unification of the various local forms of aiding, a common work ontology is necessary. Ontology, as defined by Eggleston, "is the set of terms, meanings and relations between terms that captures or represents some subject matter" [6]. Therefore, a work domain ontology would entail the terms and meanings a worker uses to think about and accomplish work tasks. In association with this work ontology, the domain model is also expressed from the worker point of view. This

form of model is a work ecology model because of its inherent relative relation to the worker [7]. The work ecology model acts as the habitat for both representational form and software agent aiding methods. It is the foundational framework where the various forms of support blend together into a work-centered support environment. The work ecology model, in summation with both aiding methods, establishes a homogeneous and unified work support system which is efficient and effective in helping the user to accomplish work. A conceptual diagram of the WCSS interface client technology is provided in Figure 2.1.

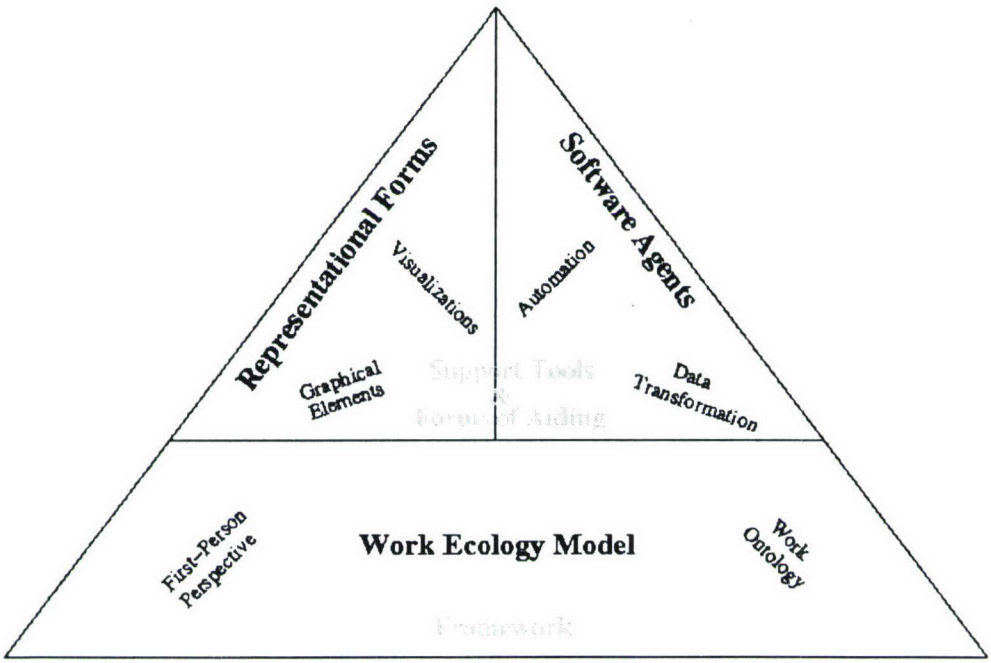


Figure 2.1 WCSS Interface Client Composition

2.1.2 WCSS Design Technology

As a design technology, WCSS requires principles, concepts, and strategies for reducing work complexity in creating a work-centered client [6]. Prerequisite to any WCSS being developed in practice, a design methodology must exist that suits the creation of work-centered software. This emerging design framework, formally labeled Work-Centered Design (WCD), illustrates and emphasizes important features

and concepts of a work-oriented design. Eggleston states that a WCD framework is, "consistent with, and in many ways, overlaps other methods to design that are known in broad terms as the cognitive engineering approach to human-centered design" [9].

A key tenet in the success of WCD is the notion of coherence. The sustainability of vital work-centered details must occur from elicitation and knowledge capture to implementation and final development to certify a successful WCSS. Without the potential to communicate how requirements and design intent interconnect, the probability of creating a stable, adaptable, and coherent WCSS is significantly diminished. Transmission of fundamental design artifacts is indispensable in order to prevent the destruction of the designed work-centered orientation at development trade-off points. As a new design framework technology, WCD continues to mature towards a fully coherent framework.

Work-Centered Design is carried out with very close ties to the work ontology as mentioned in section 2.1.1. This set of terms, meanings, and relations on the contextual subject matter are made visible in a WCSS through Work-Centered Design [6]. In order for this to happen, the designers must have a deep understanding of the cognitive and collaborative demands of the work domain [8]. There are three principles which stand out and are used extensively during the process of WCD. These principles are: the First-Person Perspective Principle, the Focus Periphery Organization Principle, and the Problem-Vantage-Frame Principle. These principles represent the building blocks of WCSS development.

The First-Person Perspective Principle is the core element of the work-centered approach to design a "work representation in terms of how workers see and engage work" [4]. This means the worker's ontology should be used as the primary

vehicle for describing any and all visual interface components. This relieves the user from needing to "interpret" the software in order to comprehend how it corresponds to the actual work environment Not only this, but the First-Person Perspective is also sensitive to the manner in which a worker engages and completes multi-part work tasks. In this way, not only individual visual screens provide a work-oriented aiding mechanism, but also the collaboration of the entire visual package does so by behaving in a logical or sequential format which follows that of the contextual work tasks. By mirroring the patterns in which the user performs actions and events, the support system reduces cognitive and procedural burdens on the user.

The Focus-Periphery Organization Principle was developed as a result of identifying design patterns recurrent to WCSS interface designs. The theme of a central frame focus has become a canonical element of all WCSS designs to date [4]. Non-focal factors which are essential to decision making, yet are not among the most crucial features, are relegated to the periphery surrounding the central frame. Through this combination of center and periphery, the entire referential context can be preserved in the viewing client, yet an order of importance is still maintained to aid interpretation and data retrieval.

The Problem-Vantage-Frame Principle addresses the nature of work tasks as an unfolding series of problem solving events [4]. Each individual problem event which must be completed as part of work exercises is specifically identified in order to attune the interface to all relevant factors pertaining to decision making and operations. By doing so, the interface can be designed to encapsulate the referential coordinates, level of detail, and level of abstraction appropriate for specific work domain variables [4]. With this in mind, the goal of the overall interface is to accommodate the vantage point (or vantage points, as typically there are many in a

single WCSS) which a user may adopt to meet the current situation [4]. In effect, this design strategy moves logically from problem to vantage, and then to the final instantiated interface frame.

The WCD framework coordinates having a first-person perspective with the current work domain context to ensure the interface system aids the worker in completing their responsibilities [9]. An overview of the current WCD framework is presented in Figure 2.2.

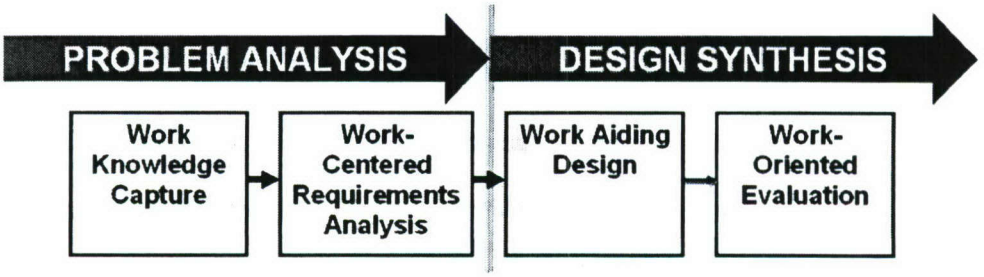


Figure 2.2 Work-Centered Design Framework

The initial stage of WCD is known as work knowledge capture. In this stage, the focus is on capturing knowledge about the work system's organization [9]. All of the goals during this stage revolve around familiarization, understanding, and discovery of the worker, work context, and work practices. During this process, information may be collected out of a broader context than simply that of the system to be developed. Doing so collects necessary details about the exterior work context in which the system will be placed. The information acquisition done in this phase builds a knowledge base of the richness and complexity of the work and work context that can then be used by the designers to build a work-centered support system [9].

The next stage in the framework for WCD is work-centered requirements analysis. Here, the captured knowledge is further analyzed to draw out properties of work in which the customer's requirements are embedded [9]. The goal here is to

separate aspects of work into various categories which are more naturally and logically partitioned [9]. Among these different types of design requirements are: functional, informational, decision making/problem solving, and situational awareness requirements [9]. This elicitation of work requirements is technology independent and states requirements in a succinct understandable manner.

The work aiding design phase of WCD is central in the aim to create a work-centered product. Proceeding from the requirements taken from the first two steps, an analysis is made from a cognitive and human factors engineering perspective before a work-centered design of the system can be first conceived. The various cognitive requirements which have been collected lead and influence how the design will be constructed. An area which has been much neglected in the past, the work aiding design phase preserves the work-centered requirements in the form of the design before being passed onward for final implementation. Specifically this includes determining what forms of direct and indirect aiding can be used, as well as how elements of the workspace context can be represented clearly and effectively. These types of factors come together in the expression of a design that has the worker's first person orientation at its center. At the end of this phase of design, screens and prototypes are commonplace, displaying the appearance of the set of work-centered requirements. However, as this phase seeks to explore some of the "uncharted territory" of designing systems which are cognizant and work-aiding, the set of tools and forms to support the communication of said design are grossly underdeveloped. This includes the transfer of the final work-centered design synthesis to software engineers and developers for final implementation. This issue is often intensified by the fact that software personnel rarely have knowledge of the actual work context from which the design was developed. Without a medium for communication in this

final active step of the WCD process, the entire work-centered design is at risk of being made ineffectual toward the end product.

Finally, a work-oriented evaluation is done to assess usability, usefulness, and impact of each design prototype. This can include non-traditional measurement techniques to ensure that the design meets the cognitive requirements set out from analysis. As a new design technology, the Work-Centered Design framework continues to be improved and refined. A complete and definitive evaluation method is yet to be completed, thus this phase at present is often a conglomeration of various techniques which can be used to verify design principles.

2.2 Software Requirements

2.2.1 Requirements Description

In order to establish an accurate depiction of software behavior, requirements are drafted to describe essential constraints. While the word "requirement" is used quite commonly among software professionals and related circles, its meaning is usually subjective. What form and structure requirements should take and how they should be written varies greatly depending on the consulted sources. Sometimes requirements are viewed as being very high-level, abstract views of the needs of a system. Other times, requirements are seen as the concrete, unchangeable formal definitions of system functionality. Without recognizing that both definitions are acceptable in certain situations, it is unwise to describe a certain style or requirement definition as universally adopted.

Software requirements are often grouped into categories relating to certain shared characteristics, features, and details. These divisions are logical separations to

call attention to the differences between two specific types of requirements. When discussing these sorts of requirements classifications, it is important to recognize that these categories are not always mutually exclusive. For example, a requirement that is classified as a functional requirement may also turn out to be a domain requirement. A quality assurance requirement may also be a security requirement. The ability for a requirement to be analyzed and categorized into more than one logical group opens the door to a world of confusion. If a development team is not uniform in their understanding and usage of the various classifications, project development woes will likely ensue.

Creating a thorough and reliable specification for all of the various unique requirements of a software system is quite difficult to achieve in the common field of practice. Specification SRS documents often serve as a contract between contractor and client as to what is expected of the final resulting system. But, there must be agreement on what constitutes a satisfactory software requirement. Both parties are interested in a project's overall success. Therefore, it is common that large amounts of collaboration take place before an agreement is made on an SRS. Although there is no standardized format in which the requirements are stated, there are commonly used criteria that have been duly noted and propagated by IEEE and ISO. Taking a glance at some of these criteria listed in IEEE 830 and ISO 9126, [10] gives the following listing:

- **Correct** each requirement is an accurate depiction of what the client needs in the final solution.
- **Complete** there are no extra details or features of importance which are left outside of the requirement
- **Unambiguous** there should be only one interpretation of a requirement.

Being able to extract a meaning other than the one intended should be suppressed as much as possible.

- **Consistent** there should not be conflicting requirements. The requirements should fit together to partition the entire system space.
- **Ranked** in terms of importance and stability, more important requirements should be shown to be of higher priority. Unstable requirements should be fully explained to understand risks associated with them.
- **Modifiable** a specification should be as easy to change as possible. Changes during development can and do occur often.
- **Verifiable** each requirement should be able to be verified later in development to assure that it was indeed satisfied.
- **Traceable** a requirement should be able to be followed starting from elicitation and design through implementation and completion. No requirement should be left outside each phase of development.

This listing is by no means exhaustive in covering what characteristics help to ensure that a piece of software will meet all the wanted requirements, but it provides a stable reference point. Many items in this list can be very subjective in nature, again causing issues of miscommunication to be possible pitfalls. However, by applying these principles as much as possible, a software requirements document can progress towards a more uniform and thorough outcome.

2.2.2 Functional Requirements

Functional software requirements are detailed statements about the services the system should provide [1]. They are explicit in instructing how the system will behave to specific input and actions. During requirements engineering, the set of

functional requirements is elicited and established. These requirements form the basis for expectations between client and developer concerning what the final product will entail. Therefore, it is important that functional requirements be stated as precisely and accurately as possible. Imprecision or changes to the original functional requirements given can severely stymie development efforts, contributing to many software engineering problems [1]. Although these requirements can be fairly abstract in nature, they should attempt to follow the IEEE 830 criteria as much as possible. This can alleviate the occurrence of costly mistakes and omissions.

2.2.3 Non-functional Requirements

Non-functional requirements, as their name implies, are requirements which do not specify functions which the system should perform. Although the title "non-functional" may make these requirements sound less important, this is far from the truth. On the other hand, these types of requirements can be just as essential, if not more so than the functional requirements. Depending on the circumstances and context for final software deployment, a software effort can turn into a failure due to the omission of non-functional requirements. Non-functional requirements specify "how" the system is to perform functionality [10]. These vital constraints on emergent system properties include areas such as: security, ethics, reliability, maintenance, response time, availability, and usability. The difficulty that comes with many of these requirements is the inability to specify them. Engineers often note that non-functional requirements are critically important, but that they do not have any way to specify them, and little help to do so is available [10]. As a result there are often widely varying methods of stating and communicating non-functional requirements during the software process.

2.2.4 User Interface Requirements

One major subject of attention in the non-functional requirement realm is user interface (UI) development. The UI is the main access point through which the user interacts with and makes use of the software product. In order for the software to be utilized in an effective and efficient manner, the user interface must be accessible enough that the client has no major hindrances in employing it. This milestone usability requirement has been the focus of many development efforts. How do you ensure that the user interface will incorporate good human factors design? How does the UI fit together with the functional backbone of the software? The incorporation of the user interface with the rest of the software system causes a collision between the functional and non-functional requirements. Yet, this aggregation is inevitable in the development of high-quality useful software.

2.2.5 Unified Modeling Language as a Requirements Modeling Tool

The Unified Modeling Language, or UML, is arguably the most successful and effective standard modeling tool in the past decade of software engineering. The UML serves to piece together the scattered details of design into a coherent standard model which can be used to power the software process towards completion. It has become a reliable and robust tool in communicating intent between client and contractor, and a development hub for documenting and incrementing changes to original design plans. The UML is very well suited for functional requirements. Its multiple diagrams enhance the number of views and interactions that can be displayed. This allows system behavior to be well modeled and understood prior to the implementation of the product. However, the UML does not accomplish what is lacking in the non-functional requirements focus. While being generally flexible, the

UML does not have a direct answer to how a UI should be designed, implemented, and integrated with the remainder of the software content.

2.2.6 User Interface Development

The manner in which a user interface is instantiated normally ranges dependent upon development context. Sometimes the UI is completely developed in-house by the same team who design and implement the rest of the project. Other times, a team of domain specialists and human factors engineers are involved to aid in the creation of an effective UI. The methods used within this process also tend to vary. Diagrams, prototypes, use cases, and scenarios all serve to display how the interface will respond and behave under certain circumstances. These tools may work well to design and modify the UI itself, but they are not sufficient to integrate the UI development with the functional development at large.

In recent years, it has been noted that large numbers of software systems are not very effective due to poor usability. As this trend has advanced, more of a focus has been placed upon usage during software design and production. Under the umbrella of usage-centered design, more of an emphasis is placed on the UI development and how it relates to the rest of the software. The software development paradigm continues to shift away from user-centered towards usage, implying that the amount of usability in a system is in direct relation to how effective it will be in the field of practice. This assertion represents the positive thrust from software engineering toward cognitive and human factors engineering. In order for today's complex software systems to be truly capable, they must be made more contextually relevant. By supporting the client's work and work environment through user interface interaction, more attention can be given to the tasks at hand, rather than to

the details of manipulating the software. This is precisely the goal which the work-centered support system methodology pursues.

3. WORK-CENTERED SUPPORT SYSTEMS IN PRACTICE

As the WCSS design ideals and terminology have evolved, several test developments have been done to further initiate discoveries and progress in the maturation of WCSS theory. Each of these developed WCSSs has been accomplished through the Air Force Research Laboratory (AFRL) in direct association with those developing the WCSS model. The resultant products have served dual-purposes in aiding the progression of WCSS design theory, as well as achieving a real life software solution to a problem facing the United States Air Force (USAF). Each of these systems merits the framework and details described in sections 2.1.1 and 2.1.2. In this chapter, information on the Work-centered Interface Distributed Environment (WIDE) project is covered in detail. Two other WCSSs are briefly presented to further illustrate work-centered concepts.

3.1 Work-centered Interface Distributed Environment (WIDE)

The Work-centered Interface Distributed Environment is an ongoing WCSS project being developed to provide advanced human-computer interfaces to plan and monitor Command and Control (C2) missions. The WIDE project is in direct association with the Air Mobility Command (AMC) operations center for centralized command and control and the Tanker Airlift Control Center (TACC). The TACC is a

global air operations center with hundreds of people planning, scheduling, and tracking about 350 strategic tanker and airlift missions per day [11]. By their very nature, airlift missions are both dynamic and complex. The task of planning a mission involves dozens of factors related to distinct individual sources. As missions themselves are quite variable, the job of planning and re-planning them is correspondingly dynamic. A trained mission planning team has the job of coordinating all the various pertinent mission-related data and communicating with the respective parties involved. These relevant activities include: matching loads and cargo to available aircraft, diplomatic clearances for landings in and over-flights of foreign nations, airfield and airspace constraints, air refueling constraints, and others [11]. Aside from mission planning, much effort and time is also put into mission execution. It is often not until a few hours before a mission is launched that it can be evaluated for adequacy and feasibility [11]. The process of mission execution includes many extraneous tasks to that of mission planning including: finalizing various information and flight plans, obtaining appropriate clearances, receiving appropriate permissions, and coordinating other mission-vital details. Overall, the C2 work tasks are numerous, situation-specific, and interrelated in complicated ways [11].

In order to effectively carry out their responsibilities, the mission planning team must multi-task between mission-related activities such as: monitoring, re-planning, analyzing, computing, predicting, and communicating. This variety of work tasks can easily become quite burdensome when dealing with real-time requirements, exceptions, delays, and personnel. Unfortunately for the staff team, the current computer systems used to support and manage the execution of the TACC are legacy, data-centric systems [11]. Critical mission data is shown on a variety of separate display panels and is not always delivered at the right time or in the right format for

mission planners to be able to make effective decisions. This means that duty officers must piece together data from various places and then often do complex computations intuitively before any true decision can be made and carried out. When conflicts occur within airlift missions due to changing real time factors, mission planning officers must first discover the referring issue by constant monitoring. Then after discovering such an issue, a duty officer must navigate different information panels to locate the exact nature and context of the alerted problem. Thus the mission planning team is put under a large amount of unnecessary cognitive and managerial burden because of the inability of the current systems to provide effective support for their work.

The WIDE project attempts to address these concerns by supporting the cognitive aspects of work through a unique blend of visualization and automation, cognitive work-aids, and human-computer upgrades to current C2 systems. By studying the work context for the C2 systems, a suitable solution which captures the nature of the work itself was designed. This design will help mission planning officers in numerous practical ways by improving their situational awareness of the various pertinent mission factors, as well as improving decision quality by displaying information in a timely and more accurate format for analysis. The effect of WIDE being developed and integrated into existing C2 systems will mean better planning and monitoring of missions, easier recognition and response to problems, and less difficulty in the management of multiple ongoing missions.

3.1.1 WIDE Spiral One

WIDE is being developed in a series of three progressive spirals, each encompassing a different portion of the overall distributed environment. The first spiral is the foundation of the WCSS. Its concepts will deal in particular with the

development of cognitive work support visualizations to display the mission timeline and related views. The major development component of spiral one is the Timeline Tool mission display. The Timeline Tool will replace the legacy systems which currently cause the process of mission planning to be so burdening and complex. As a complete mission planning and monitoring software package, the Timeline Tool will be the focus of WIDE spiral one.

3.1.2 Timeline Tool WCSS

The Timeline Tool is a mission planning WCSS, built to aid aircraft scheduling and operations. It aims to reduce the number of errors committed during mission re-planning, help recognize the impact of mission-related decisions, and lower the overall response time in dealing with mission alerts. As such, the user interface for the Timeline Tool software can be expected to be a complex aggregation of a large amount of data into a suitable, work-aiding form. The Timeline Tool user interface can be divided into two major views, each with a specific set of important requirements. These two views are the multi-mission timeline display and the detailed mission timeline display. Both views are oriented along a horizontal axis correlated to time, a key feature of the software as the overall name implies. All figures and examples given in this section are taken from the actual design drawings created for the Timeline Tool [12].

The multi-mission timeline display provides an overview of all missions within the current timeframe. This view serves as the home screen for the Timeline Tool and allows officers to view core details about many missions simultaneously. Having an outer vantage point increases the duty officer's situational awareness in being able to monitor many missions from a single viewing screen. Streaming data is

taken in via the Timeline Tool's communication links and each mission within the multiple view is updated continuously. When an alert is raised due to delays or any number of other factors, the specific mission in question will signal an alert to call the attention of the officer on duty. The officer may then refer to the detailed mission view of the alerted mission in order to ascertain the problem. From this outer panorama, monitoring and responding to circumstances within individual missions is made an easy task inside the multi-mission view.

The multi-mission view screen itself can be broken down into a series of component interface areas. Figure 3.1 shows a design image of the multi-mission display.

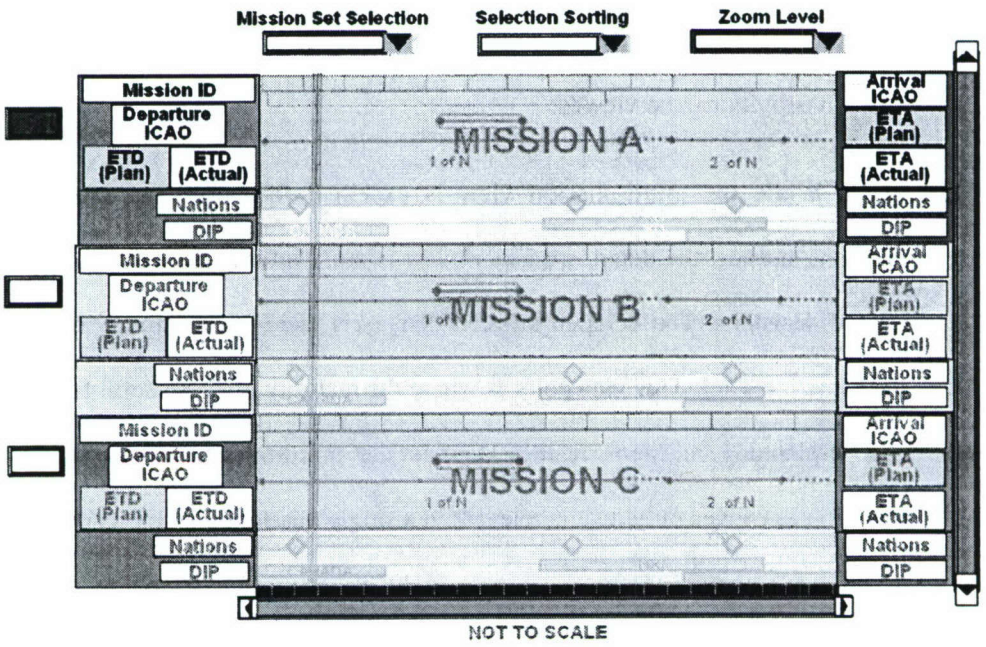


Figure 3.1 Design of Multi-Mission View, used by permission [12]

At the top of the multi-mission view are utilities which allow the user to easily sort or select criteria for viewing a certain set of missions. As a large number of missions execute concurrently, these features make it easier to monitor specific mission types. On the left side of the design screen are a series of buttons which,

when selected, activate the detailed mission view for the corresponding mission. Switching between the multi-mission view and the detailed mission view is therefore a simple navigation. Within the center of the multi-mission view is a collection of what are referred to as mission cores. The mission core constitutes the key features and information about a mission which distinguish it from all others. The mission core will be covered in detail within the explanation of the detailed mission view, but for now it is worth noting that it is composed of a main viewing display surrounded by two peripheral sidebars. The Timeline Tool adjusts its display window by default to the current time of day. The worker is provided with a set of scroll bars (both vertical and horizontal) to allow traversal of mission core data and time display interval. Using the vertical scroll bar, the entire set of missions can be accessed. Using the horizontal scroll bar, information regarding completed past missions and upcoming mission activity can be viewed.

While the multi-mission view is excellent for observing details of many missions at once, the detailed mission view is more informative for making mission-related decisions. The detailed mission display is therefore a primary component of the Timeline Tool. This viewpoint is where the majority of data useful for monitoring and re-planning missions resides. Unlike the multi-mission display, the detailed mission view gives only details relevant to a single mission, making it specific enough to show all pertaining factors which might affect mission planning activities. An image of the detailed mission display is given in Figure 3.2.

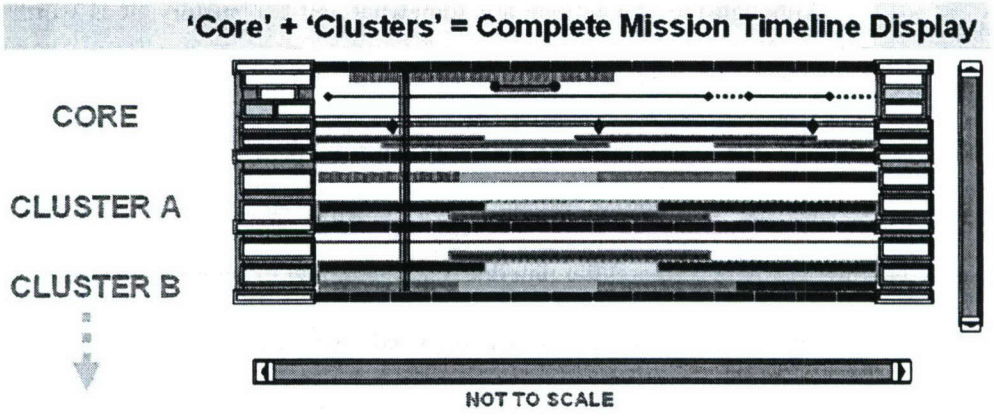


Figure 3.2 Design of Detailed Mission View, used by permission [12]

At the top of the detailed design screen is what is referred to as the core display panel. The core display is the heart of the entire Timeline Tool. Contained within the core are the distinguishing values (mission id, commencement and completion times, and aircraft numbers) which identify each unique mission. The more appropriate technical name for the core display is the flight data depiction. This is where the primary flight information is held. An annotated design image of the core display is provided in Figure 3.3.

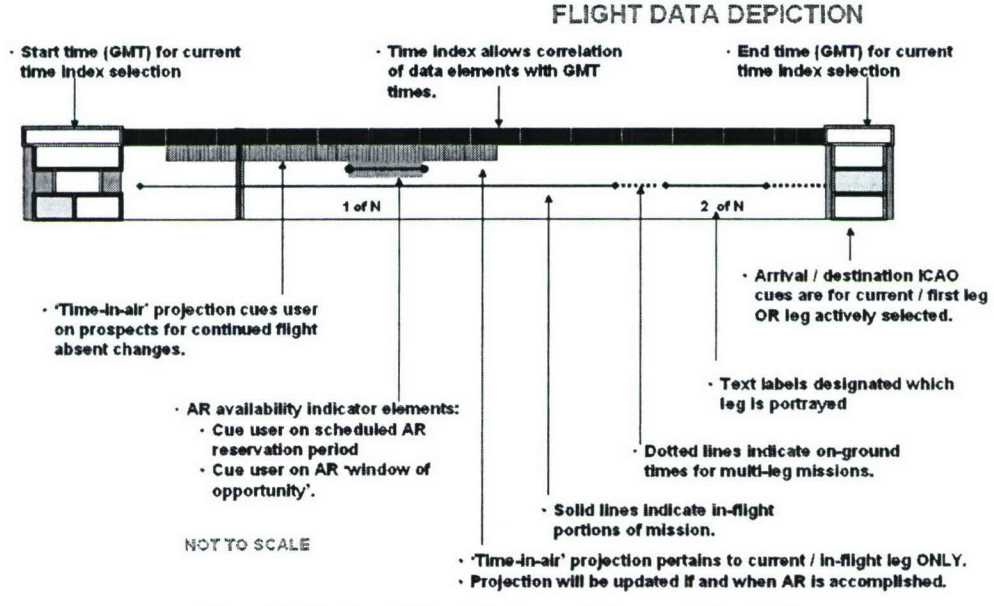


Figure 3.3 Design of Core Display, used by permission [12]

Although the annotations are somewhat self-explanatory, it is worth noting several important aspects of the core design. First, there are peripheral areas surrounding the central visual window which are the location for valuable numeric data related to the mission, flights, identifications, etc. Second, the central visual window is where the essential timeline data pertaining to flights, air refueling, and all other flight related actualities are depicted. The horizontal formatting of this information is crucial to the overall work-centered orientation as it allows a user to plot all aspects of the mission along the horizontal time axis.

Directly underneath the flight data depiction are a series of mission-related data areas, organized into separate visual clusters. These clusters all share the same horizontal orientation of the core display. This makes the entire perspective of the detailed mission view a consistent work-centered one, catering to the mission planner's need to see all concurrent activities in a way which aids sense-making and decision-making priorities. Each cluster contains a distinct category of flight information. The scalable layout of the detailed mission view allows the addition of an arbitrary amount of clusters. This extensibility attribute may be exploited in future upgrades to the Timeline Tool. As of this writing, the following clusters have been created: geographical features, port (airfield), aircrew, aircraft, ground events, load/cargo, and diplomatic permissions. As these clusters are very close in visual structure but differ in actual information, it is superfluous to go into the details of each one individually. For the purposes of this paper, the port (airfield) cluster and the diplomatic permissions cluster will be used as representatives of the entire cluster space. Design images of each of these clusters are provided in Figures 3.4 and 3.5.

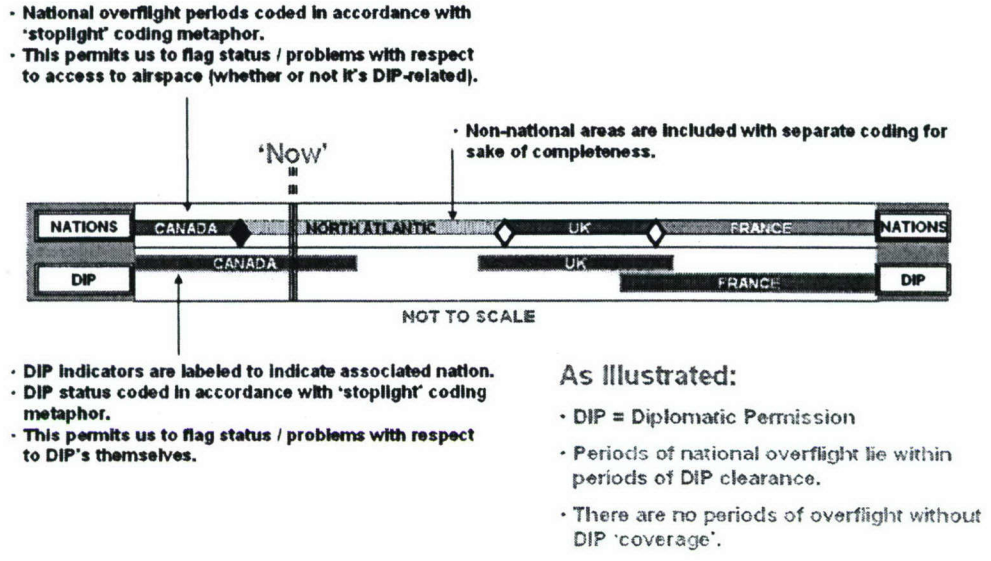
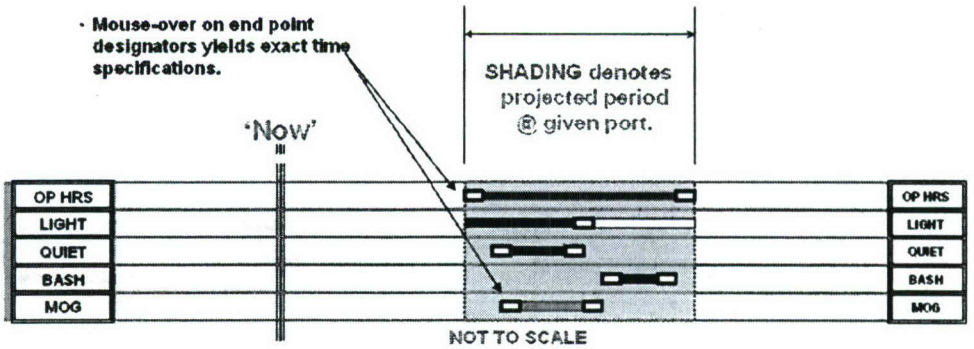


Figure 3.4 Design of Diplomatic Permissions Cluster, used by permission [12]

The diplomatic permissions cluster (or DIP cluster) shows information regarding nations which are being traversed during a mission. An aircraft must obtain a corresponding diplomatic permission in order to cross foreign airspace. This cluster displays which national boundaries will be crossed and how long diplomatic clearances have been obtained, all with respect to time. Gaps in coverage are made much easier to spot visually using this orientation rather than relying on numerical time segments and off-hand mathematical calculations. As with the core display and all other clusters, peripheral data is located in boxes which flank the central visual display.



NOTES:

- **Unlike other clusters, Port elements are portrayed only for period during which the mission is projected to be at the given airfield.**
- **Because of this, the Port elements are the only ones which are 'horizontally constrained' (with respect to representation).**
- **This tactic is necessary to constrain visualization to reflect only the pertinent factors.**

Figure 3.5 Design of Port (Airfield) Cluster, used by permission [12]

Generally, the port cluster follows the same style guidelines as that of the core display and other clusters. Distinct, however, to this cluster is that elements regarding specific airfields available during a mission are only to be displayed when the aircraft is in range of the airfield. This minimizes the overall amount of data on the screen for this cluster, thus reducing burden to the officer, who only must analyze the port data when it is relevant to a specific mission. This constraint is interesting as it shows that while each cluster is quite similar to the others, they all do maintain not only individual data, but also individual display requirements.

The Timeline Tool also contains various aspects of automation, which directly assist the user in performing calculations and simulations during usage. These automating agents range in complexity from simple time difference computations to on-the-fly repositioning calculations. Many of these facilities are associated with the simulation mode feature of the Timeline Tool. Having a simulation option within the tool enables a user to directly interact with the data and planning components of a mission rather than attempting to perform difficult mental projections. From a visual

perspective, nothing changes when the simulation mode is activated (aside from an indicator that the user is in simulation mode). The user is then able to click, drag, and reposition core and cluster elements to view the effects they will have on the overall mission. In this way, a worker is able to re-plan missions in a simulation context before contacting air personnel and giving guidance. While in simulation, the user can immediately view conflicts and associated risks related to whatever re-planning is being simulated. Automation agents handle the functional details associated with these operations and alert the user accordingly. Using automation facilities to do complex predictions and potential forecasts is an immense cognitive burden relief. The result is fewer mistakes made due to inadequate planning tools. The majority of this portion of the WCSS is seen only through the informative alerts given by the system, as each agent runs within the functional context of the supporting software.

3.1.3 WIDE Spirals Two and Three

Spirals two and three of the WIDE project will serve to enhance the WCSS capabilities and scope developed during spiral one. Spiral two will focus primarily on mission management views and the networking of separate support tools together for versatility. This may include the fusion of tools such as the GAMAT system described in section 3.2.1. It also includes the initial integration of spirals one and two into the TACC environment. Spiral three includes development of mission team displays and TACC personnel tools. Together the three spirals will cover the many daily planning tasks that mission officers at the TACC must engage, producing a software system designed to accompany those officers in their assignment completion.

3.2 Other Related WCSS Examples

3.2.1 GAMAT WCSS for Global Weather Management (GWM)

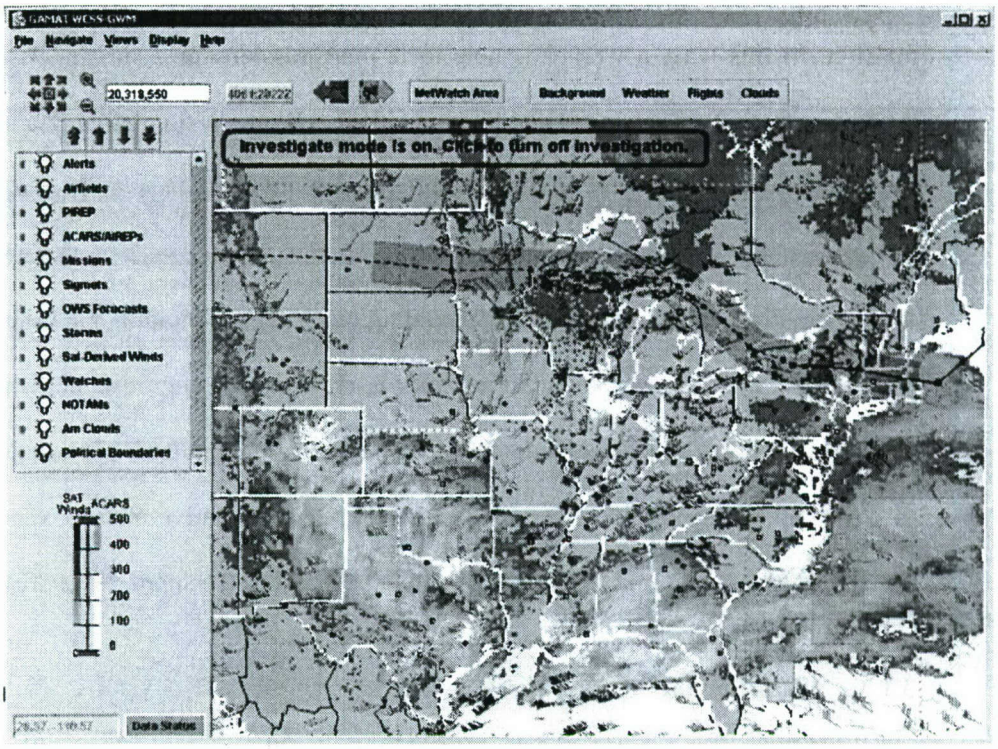


Figure 3.6 GAMAT WCSS for Global Weather Management

The WCSS for Global Weather Management known as GAMAT is another software system developed and implemented by the Department of Defense and the AFRL. GAMAT's visual interface has several noteworthy characteristics which differ from those of the Timeline Tool in section 3.1.2. GAMAT is highly focused upon a central visual weather imagery screen, rather than a timeline. This screen depicts multiple layers of interrelated geo-spatial data. As a work-aiding support, GAMAT allows the user to have strict control over what layers of weather data are shown on a particular map image. Using a combination of colors, textures, and graphics, the central imagery screen keeps the user contextually informed regarding current weather patterns. Sortie data is imported into the WCSS in the form of a sortie palette,

in order to compare and relate it to weather data. The sortie palette acts as a companion to the central imagery screen. GAMAT also contains extensive navigation tools and automation agents to create user created "watch areas." As a whole, the design and support structure of GAMAT differs from that of the Timeline Tool due to its central focus on weather images and companion sortie palette. In order to effectively manage media such as weather imagery screens, the interface client must be adapted quite differently than it would for standard numerical data streams such as those the Timeline Tool uses.

3.2.2 The Coronet Awareness and Team Synchronization (CATS) Project

The CATS project is being developed as a work product of the Work Support Research and Development (WSRD) program. This program exists to develop and deploy WCSSs into various United States Air Force (USAF) systems. The CATS project will be a net-centric application designed directly to match Coronet needs. Coronet missions are missions involving the movement of aircraft, cargo, and passengers from one place to another on long, trans-oceanic trips. Each Coronet mission leg consists of receivers, which are smaller aircraft with small fuel tanks, and tankers, which are larger air-refueling aircraft. The CATS application will possess both single and multiple leg panels. As a WCSS, the CATS project will be a challenge, as each mission leg has different user viewpoints for accomplishing different tasks. This means that a solution WCSS will need to match the vantage and focus of the user for each mission segment in order to establish an overall work-orientation.

4. VISUAL REQUIREMENTS SPECIFICATION LANGUAGE

4.1 Language Basis

4.1.1 Overview

Creating a specification language that is able to encapsulate the important aspects of the visual design portion of a WCSS will allow coherent work-centered software systems to be created. Such a specification creates a framework for the transmission of visual aspects of design in a precise and unambiguous fashion. Additionally, such a language enables flexibility and concision in the selection and labeling of explicit design elements. Most importantly however, a specification language for visual requirements would have a wide variety of contributing application such as: being used by cognitive engineers and user interface designers as a work-centered design tool, providing a format for capturing visual non-functional requirements, and supporting an area of modeling not provided by the UML. Such a language would empower those who are most familiar with the work-centered design, to convey the important design artifacts to software engineers. To a certain degree, a visual requirements language would have the same purpose and goal as that of the UML: to provide a medium of communication to talk about software system modeling, although the focus becomes somewhat more specific. In this chapter, a visual requirements specification language is proposed and described as a solution to the plaguing problems of coherently producing WCSS software.

In order for a specification language for visual requirements to be truly beneficial, it must integrate well with existing software engineering modeling techniques. Using a language as a standalone requirements specification amid other gross functional requirements and modeling documents will only increase its chances of being overlooked during development. An effective modeling document is one in which the system design can be best represented in its entirety, as not to additionally burden the development team when they proceed to build the software. This motif of having everything in one place thus implies that a specification language for visual requirements should integrate well into a standard modeling language such as the UML. The UML makes this quite feasible through its natural extensibility. Using the UML's existing outlets for connecting outside modeling techniques will provide the needed linkage for a visual requirements specification. This advantage opens further possibilities for the specification language towards integration into a complete work-centered software process. Further discussion and details regarding UML and process integration are saved for Chapter 6.

4.1.2 Specifying Work-Centered Visual Requirements

As stated in section 2.2.3, it is often the non-functional requirements which are the most difficult to quantify during software development. Due to their often qualitative nature, measuring whether a non-functional requirement is being addressed and met in the product to be developed is a challenging problem. This is an issue that the software community commonly faces and must deal with appropriately in the creation of any software with such requirements. The best solution in many cases is to use requirement-specific methods of both validation and verification. These metrics ensure that particularly important requirements are as correct as possible. Without any way to follow non-functional requirements from inception to completed product, there

is high potential for unsatisfactory results. Such results can spell disaster for an entire project and are to be avoided at all costs.

In a WCSS environment, the work support is directly and intrinsically tied to the user interface visual display. It is in this contour that both the representational forms and automation agents combine to form a user support system. Consequently, the on screen visual display is crucial to the overall success of the resulting WCSS. Visual requirements, such as how a user interface looks and behaves, are typically qualitative in nature, making them difficult to declare and convey. In past WCSS development, a series of images has been given to the development team along with excessive prose instructions of select visual details. This format is hardly suitable to achieve coherency in the overall development process. Selected notes, explanations, and details scattered indiscriminately across many documents and diagrams make accurate development an arduous challenge. The visual requirements specification language aims to fill a role which will lessen this burden considerably if not completely. By creating a framework in which important visual requirements and UI details can be captured in a semi-formal manner, the language can serve as a precedent for the communication of work-centered concepts across the development lifecycle.

Developing a language to capture a set of user interface requirements is a necessarily difficult task. User interfaces in and of themselves are complex, dynamic, varied, and often contain many subtleties. A specification language faces the difficulties of needing to achieve a wide variety of functions in order to be successful. Summarized below is a list of various component attributes that must be present in a good specification language.

- **flexible** A language must be able to scale well to the overall volume of requirements, whether large or small. It must also be able to capture minute details if necessary. By covering a wide variety of aspects and features which are characteristic of UIs, a good language should be able to cater to any UI, not just certain types.
- **precise** A language should remain as unambiguous as possible, as to avoid errors in interpretation during development. Certain requirements should not be less ably specified than others.
- **clear** A language should be as comprehensible to both UI design experts and software engineers as possible. Since it is being employed to transfer valuable design information, it must communicate data reliably. This applies to both structure and format.
- **integrable** Being able to integrate well with existing software engineering methods is an important aspect for a language to maintain its usefulness. If a language becomes another stand-alone method, it greatly reduces its chances of being meaningfully employed in a real world setting.
- **augmentative** A language should contain a facet for emerging new trends in UI development. This allows the language to be used for new and future design techniques.

This list of attributes is not exhaustive, but provides the groundwork which a language must cover. In looking for existing modeling languages which might support these many features, the web development field provides many plausible choices. Of particular interest in this case is the increasingly growing market of Extensible Markup (XML) Languages. Amid the many possibilities for developing a new

specification language, XML-compliant languages possess an exceptional amount of potential.

4.1.3 Extensible Markup (XML) Languages

The Extensible Markup Language (XML) was designed and released during the late 1990's, but has found a growing amount of usage in more recent years due to high content demands on the World Wide Web [13]. The XML is similar to HTML (Hypertext Markup Language), with the main difference being that the XML can be completely configured to better annotate and represent specific application features and content [13]. This separation from being a strictly structural language allows a myriad of XML languages to be created using the XML basis, yet pertaining to specific application domains. Among the design goals for the XML are the support of a wide variety of applications, ease of use, and comprehensibility [13]. However, each new language which is rooted in the XML still retains its foundational structure of tags and nesting. The XML also contains simplistic, yet effective cross-referencing mechanisms for linking various elements. As many more XML compliant vocabularies and meta data languages are being created, developers are finding it easier to create markup for distinct application domains and take advantage of the XML's parse-able formatting [13].

4.1.4 User Interface Markup Language (UIML)

The User Interface Markup Language (UIML) is an XML compliant meta-language for describing user interfaces. Its documentation reads, "the design objective for the UIML is to provide a canonical representation of any UI suitable for mapping to existing languages," [14]. This canonical representation is quite useful, as UIs are created using a large variety of different host languages. The UIML also complements

the object-oriented view, placing it in-line with current UI design practices. Typically, the UIML is used to describe generic window-based user interfaces and is then passed into an interpreter which implements the design into a higher-level programming language such as Java or C++. Unfortunately, these simplistic designs are not frequently applicable to unique and complex UIs such as those of a WCSS. However, the canonical representation still possesses many advantages in specifying a more intricate user interface. By highlighting the versatile features of the UIML, a new specification language can be drafted to capture more complex interface designs with the goal of transferring them coherently to a human development team instead of to a machine interpreter. These enhanced visual markup designs serve as a visual software model which can then be incorporated with other design documents to provide a uniform design model. An example illustration of the UIML's employment is given in Figure 4.1.

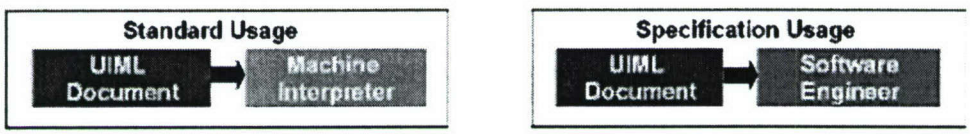


Figure 4.1 Illustration of UIML Usage

A visual requirements design model using UIML as a basis would connect the overall software process in the areas of UI modeling and visual requirements. The set of modifications which transform the UIML into a visual requirements specification language are described in this section. Beforehand, it is worthwhile to mention several useful features and attributes of the UIML as a whole before delving into its component modules.

The UIML shows usefulness for specification in its decentralized and scalable structure. As with all XML-compliant languages, the UIML follows a basic tag nesting structure. When employed, the UIML divides a UI design into a set of unique

logical parts or objects using its tag nesting structure. This enables a designer to break down a custom UI and specify each part at an appropriate level of detail. Each piece is able to contain a set of nested child parts, which can then be labeled and specified within the context of their parent. The specification proceeds recursively in this outline format until all the required details at the lowest level are captured. In association with each part's specification, the UIML uses a toolkit vocabulary to keep track of the various part types. This vocabulary is established by the designer to effectively label each class of part for correct identification. In its original context, the UIML toolkit vocabulary would provide a mapping of logical interface parts to specific class constructs in a higher-level language. For example, if a UIML design was to be implemented using Java, each UIML part would correspond to a Java object class. In the modifications proposed, this language mapping is circumvented towards a better usage for specification purposes. Incidentally, the toolkit vocabulary remains a valuable tool as a means to correctly interpret a UIML design. Using the ability to scale and effectively decentralize design requirements as well as annotate how they are to be interpreted, a designer can define unique user interfaces with precision and clarity at the individual part level in order to maintain a work-centered domain focus.

4.2 Visual Requirements Language Framework

In its current form, the UIML is a suitable basis for a specification language, but still lacks several important characteristics to make it effectual for WCSS visual requirements. First, the UIML's features for part description and definition must be made comprehensive enough to merit a clear specification of complex design artifacts. Currently, details of specific part behavior are addressed at the group level rather than as individual parts. This deficiency is addressed by the addition of attribute

tags which replace and extend the current part tag categories. Along with this, the ability to specify dynamic attributes (such as rules, conditions, and other behavior) must be further extended to handle more complex actions (such as streaming data, periodic updates, and computations). Second, since the language will be employed as a human-to-human software design protocol rather than a human-to-machine interpretation, a more understandable formatting is beneficial. In order for human factors scientists and other user interface specialists to effectively use the proposed language, it must be represented in a more suitable syntax. These issues and other minor additions are achieved by modifying the UIML to create a new visual requirements specification language. By modifying the existing UIML, we can make use of its features for formal syntax, canonical form, decentralized and scalable structure, and part vocabulary toward the goal of conveying a UI design to a development team coherently.

4.2.1 Original UIML Syntax

The original UIML syntax is composed of four major categories used to describe each part of the user interface. The four category tags are: structure, style, content, and behavior. Each part is defined in terms of these four major elements. However, in the original syntax, each of these elements is applied to parts in a sequential form. For example, first, all parts are defined in terms of their structure. Next, all parts are described in terms of their style, and so on. Therefore, it is the summation of all four categories which makes up the part description in its entirety. This is no problem for a rendering machine or interpreter, but is quite difficult to comprehend mentally by simply reading the XML design code. The visual requirements specification language makes use of each of these four categories in capturing requirements. However, all of the tags have been either modified or

incorporated into new tags given to replace those of the original UIML where further specification is necessary. One immediate example of this is that the segregated group formatting of part definition is replaced by that of unified definition at the specific part level. This change makes it much easier to locate all information and requirements related to a specific part by placing them all in a single location. This addition and others equip the new language to better encapsulate each part's requirements and simplify the process of information transfer from designer to developer. In an effort to help understand the original UIML syntax and the changes proposed to it in sections 4.2.2 through 4.2.7, **Figure 4.2** gives an example portion of a standard UIML document. This example will be divided and discussed throughout the next sections.

```
<uiml>
 <structure>
  <part class="Frame1" id="JFrame">
  <part class="Label1" id="TermLabel"/>
  <part class="List1" id="TermList"/>
  <part class="Label2" id="DefnLabel"/>
  <part class="TextArea1" id="DefnArea"/>
  </part>
 </structure>
 <style>
  <property part-
   name="Frame1" name="title">Frame 1</property>
  <property part-
   name="Frame1" name="background">blue</property>
 </style>
 <behavior>
  <rule>
  <condition>
   <event class="ValueEntered" part-name="TextArea1">
  </condition>
  <action>
   <!--1-->
   <property part-
    name="Frame1" name="title">Frame 2</property>
   <!--2-->
   <property part-
    name="Frame1" name="background">red</property>
  </action>
 </rule>
 </behavior>
</uiml>
```

Figure 4.2 UIML Syntax Example

4.2.2 Structure and Part Tag Modifications

The semantics of the structure tag within the new visual requirements language coincide with those of the standard UIML. The main purposes of the structure tag are of identification and part positioning. The structure tag dictates the varying degree of part nesting present in the user interface. As such, the structure tag is important for its influence on all other tag categories. Depending on if a part is nested or not determines whether it inherits certain styles and behaviors from its parent part(s). The structure tag, therefore, upholds the tree layout of the entire visual requirements document, and determines how attributes are to be recursively applied.

In the standard UIML, the structure tag specifies the overall part layout for the entire interface. It contains a set of nested part tags specifying each individual UI piece. Each of these part tags contains a unique id name and a class name. The unique id is used as a reference for other various tags within the document. The class name serves as a link to the toolkit vocabulary which provides information for an interpreter or compiler.

For a specification language, these components are all useful. Therefore, in the new language, each part is given a unique identifying name which can then be used as a reference point. A class name is also associated with each part to properly define the part type. In view of a specification, this class definition is the primary vehicle for the developer to understand why this part of the UI exists and, in this specific case, is important in the context of a WCSS. Each class is then properly defined in the external toolkit vocabulary, providing design rationale and universal class details. Subsequent parts are similarly defined in a recursive outline format, making use of nesting to correctly identify subsumed parts. In an effort to simplify the organization of parts within the specification, the unique id is used as the initial label for each part definition. This change is mainly for formatting reasons, as it makes it much easier to locate and distinguish different parts (as opposed to seeing all parts begin with the word, "part"). A contrast between the visual requirements language structure and the UIML structure tag is given in Figures 4.3 and 4.4.

```
<structure>
 <part class="Frame1" id="JFrame">
 <part class="Label1" id="TermLabel"/>
 <part class="List1" id="TermList"/>
 <part class="Label2" id="DefnLabel"/>
 <part class="TextArea1" id="DefnArea"/>
 </part>
</structure>
```

Figure 4.3 UIML Structure and Part Tags

A)
Name:JFrame
Subsumed Parts:
AB) Name:JLabel
AC) Name:JList
AD) Name:JLabel
AE) Name:TextArea

Figure 4.4 Visual Requirements Specification Language Structure Layout

The id attribute values given to each part in the original UIML syntax are replaced by the outline headers (A, AB, etc.) for each part in Figure 4.4. Note that this simplifies the difficulty in finding unique part names for large numbers of parts, as well as allowing multiple occurrences of a similar part to be defined without confusion (side by side buttons for example). The indentation makes it easier to discover that four parts are nested within the first, as opposed to the four one-line open and close tags in the original syntax. The class name for each part is retained within the "Name" attribute. Further formatting modifications and omissions are left to be discussed more completely in section 4.2.6.

4.2.3 Style and Property Tag Modification

The style and property tags are used by the UIML to capture the presentation details of each UI part. WCSS visual requirements often contain broad and elaborate attributes and relationships, making styles and properties highly relevant for visual specification. Attributes such as location, size, and strict formatting rules contingent upon external factors, all must be specified clearly and completely. The style and property categories are quite suitable for the task of capturing this range of static content.

The UIML describes its style and property guidelines within the boundaries of the style tag. The style tag denotes anything declared within as some sort of static content. Each article of content is identified by an individual property tag, containing

specific details for that article. Each property tag describes one element of unchanging style or content for one part. The referring part is identified through its unique id given in its defining part tag. In this manner, the UIML links together parts with respective properties. As many elements of style and content as are necessary can be defined using property tags within the confines of the style tag.

The idea of capturing static content is of central importance in a visual requirements language. However, instead of relying on references to provide the connection between parts and attributes, the new language simplifies the process by defining all style and property attributes alongside the part itself. By integrating style and properties into a broader static attribute category, intricate static details can be captured without excessive labeling. No boundary is placed on the number of allowable attributes, enabling a specification to fully capture all the required static attributes. In addition to this reorganization and renaming, each static attribute is given a unique identifier as well. This id serves a similar role to that of the part id in allowing references to be made for a specific attribute. An illustration of the differences between the UIML's style and property areas and those of the new specification language are given in Figures 4.5 and 4.6.

```
<style>
  <property part-name="Frame1" name="title">Frame 1</property>
  <property part-name="Frame1" name="background">blue</property>
</style>
```

Figure 4.5 UIML Style and Property Tags

```
A)
Name:JFrame
Attributes:
   1) title = Frame 1
   2) background = blue
```

Figure 4.6 Visual Requirements Specification Language Static Attributes

4.2.4 Behavior and Content Tag Modification

The behavior tag for each part defines how it should react to specific conditions and circumstances during the life of the user interface. Information about behavior and how an interface responds is naturally an important element of a visual requirements specification. Since it is common that many WCSS interface parts have multiple aspects of behavior, an accurate method of capturing those aspects must be present in the new language.

The UIML's usage of the behavior tag varies somewhat from that of the structure and style tags. It is defined in the same nested tag format, but contains a deeper level of subsumed tags. Within the initial behavior tag, a set of rule tags are introduced. Each of these rule tags corresponds to one instance of dynamic behavior. Within each rule tag, there is a pair of corresponding condition and action tags. The condition tag contains a set of various conditions (defined as events) that correspond to a set of criteria triggering the behavior. The action tag contains a set of resulting actions (listed as property tags) to be taken whenever the condition evaluates to true. The result is that a large amount of behavior produces quite a lengthy amount of behavior tag code. The parts which are affected by each aspect of behavior are determined by the references within the property tags at the action tag level. This makes it very easy for behavior to effect multiple parts, but more strenuous to define multiple behaviors for a single part.

In conjunction with the behavior tag, the UIML's content tag gives a way to express certain load time dynamic behavior. The name "content" can be a bit misleading here, as in this case it refers to the dynamic loading of static attributes, and not simply the static attributes themselves as with the style and property tags. The content tag is useful for attributes such as display language, which would be

determined upon load time as opposed to real time. For example, when an interface is first launched, a content tag would tell the system whether to display the interface in English or in French. The relationship between the behavior tag and the content tag can be viewed as somewhat of a partnership, with the content tag taking care of load time dynamic behavior, while the behavior tag handles the rest. At present it is unclear as to how the functionality of the content tag is of worth in a visual specification language. The content tag provides the UIML with yet another facet of flexibility from platform to platform and language to language. However, this is one area where the standard behavior tag may be sufficient for specifying an interface for development, while load time attributes (such as those of the content tag) would be added later if the software were ported to another platform/language.

In any case, it benefits a visual requirements specification language to declare all dynamic attributes in one location to avoid confusion. With this in mind, a dynamic attributes category has been created to replace the existing behavior and content tags. Under this new heading, multiple specific dynamic events can be identified as sets of event/action pairs. Each event listed under a dynamic attribute constitutes a triggering condition for the corresponding action. When such an event is triggered (e.g. a button is clicked), the expression listed under the action clause will be executed (e.g. load a new page). Each event/action pair is therefore able to capture one dynamic attribute of behavior. As was the case with static attributes in section 4.2.3, it is effective to place the description of dynamic attributes at the part level rather than separately. This eliminates unduly references and also scopes the behavior to the same level as that of its part.

In order to cater to behaviors more complex than can be expressed by simple Boolean expressions, a keyword *call* is introduced into the new language. The *call*

keyword exists to create an external functional reference for modeling complex behavior. This is especially relevant for WCSSs, due to the regular occurrence of multiple systems sharing and fusing data. By introducing an external reference point, a designer can call attention to the fact that the source of data for a particular action or event is retrieved from a specific source. In this way, dynamic actions based upon events such as mouse location, time of day, or numeric computations can be observed. Samples of both the UIML behavior tag and the visual specification language's dynamic attributes are given in Figures 4.7 and 4.8.

```
<behavior>
 <rule>
  <condition>
   <event class="ValueEntered" part-name="TextArea1">
  </condition>
  <action>
   <!--1-->
   <property part-name="Frame1" name="title">Frame 2</property>
   <!--2-->
   <property part-name="Frame1" name="background">red</property>
  </action>
 </rule>
</behavior>
```

Figure 4.7 UIML Behavior Tag Syntax

```
A)
Name:JFrame
Attributes:
   a) Event: Value input into part AD
      Action: background = red
              title = Frame 2
```

Figure 4.8 Visual Requirement Specification Language Dynamic Attributes

In Figures 4.6 and 4.7, a single dynamic attribute is declared. This attribute states that when a user inputs a value into the part identified as TextArea1 (part AD), that Frame1 (part A) change its title to "Frame 2" and set its background to red. Note that using the new language requires less than half the space as that of the UIML.

Figure 4.9 shows the entire visual requirements specification for the examples in this section.

A)
Name:JFrame
Attributes:
1) title = Frame 1
2) background = blue
a) ***Event*: Value input into part AD**
***Action*: background = red**
title = Frame 2
Subsumed Parts:
AB) Name:JLabel
AC) Name:JList
AD) Name:JLabel
AE) Name:TextArea

Figure 4.9 Visual Requirements Specification Language Full Example

4.2.5 Link Keyword

In addition to the standard modules of structure and static and dynamic attributes introduced, a linking module has been added to the visual requirements language to allow a mechanism for communicating additional part information. While the existing utilities of the specification language are complete in and of themselves. There can still exist difficulty in communicating complicated design details. While the specification language may be able to express the desired capabilities and attributes semi-formally, it can still be a challenge to mentally comprehend the outcome of the design. Hence, the *link* keyword exists to shine light into otherwise gray areas of comprehension. The *link* keyword provides a way to associate any portion of the specification with other relevant material for greater coherency. The keyword can be used within any portion of a part specification; allowing parts themselves or explicit static or dynamic attributes to be additionally supplemented. Such supplemental material could include: another document, an image, a video file, an interactive prototype, or anything else that might assist the developer. The *link* keyword is added

with the assumption that specification documents will exist as electronic resources capable of being networked with other valuable media resources.

4.2.6 Formatting Modifications

In order to make the visual requirements language comprehensible, it must be visually appealing to its users. A specification document is of little to no use if it is not easily understood by a developer. In order to meet this goal of readability and clarity, formatting modifications have been made to the original UIML prose. As can be seen in the various figures in sections 4.2.2 through 4.2.4, the new specification language drops the code-like syntax of the UIML. Instead, the visual requirements language relies on headers and indentation rather than tags to symbolize logical divisions in the text. The new language's formatting is characteristically part-centric. All requirements pertaining to a specific part are found at the part definition level. Each part begins with its unique identifier and class name, followed closely by headers for its attribute types. Attributes are distinguished through the usage of indentation for easy identification. Finally, subsequent parts are defined under the heading of subsumed parts and also make use of indentation to call-out the parent/child relationship present between related parts. The visual requirements specification language captures all of its requirements using this block template for each part. A part template for the visual requirements language is provided in Figure 4.10.

PARTID)
Name: Class Name
Attributes:
  Static Attributes:
    1) attribute 1
    2) attribute 2
    3) attribute x
  Dynamic Attributes:
    a) attribute a
    b) attribute b
    c) attribute y
Subsumed Parts:
  CHILDPARTID)
  Name: Class Name

Figure 4.10 Visual Requirement Part Template

In addition to headings and indentation, the new language also benefits from other minor formatting details. Using a different unique identifying scheme for each set of outline bullets assists in keeping various articles of the specification exclusive from one another. For example, when labeling parts, upper-case letters are used. To distinguish the static dynamic attributes from both one another and the parts, numerals (1,2,3) and lower-case letters (a,b,c) are used respectively. Other minor notations such as using simply color-coding are encouraged where useful, but not required.

The resulting overall language is both more readable and easy to create using a simple word processing application. Upon completion of the specification, the UI designer can forward the documents to the functional design team who can then incorporate them into the overall project design sent to the developer. While the new language does not keep the strict formatting associated with an XML-compliant language, it does maintain a rigid overall structure. As a result, the language for visual requirements is still able to be parsed, if such a need were desired. The implications and related significance of this are discussed in Chapter 7.

4.2.7 Additional Features of UIML

The specification for the UIML given at [14] is in its third version and continues to undergo revision. As a new specification language, the language for visual requirements only elaborates upon the core values of the UIML rather than its specification as a whole. Attempting to enlarge the visual requirements language so that it effectively maps to each specific area of the UIML is both infeasible for the scope of this research, as well as likely irrelevant to this area of research. It has not been fully analyzed as to whether the remainder of the UIML specification would be useful towards specification components in a visual specification language, although it is seemingly unlikely. The UIML's extended features, such as additional attributes for category tags, platform presentation details, and others are not immediately well suited towards conveying requirements. Rather, many of these types of features within the UIML exist to allow creative design of an interface, whereas in this case all interface design is done prior to using the specification language to capture requirements. However, this assumption does not altogether eliminate the usefulness of the overall UIML specification. Further discussion and possible future research in this area are covered in Chapter 7.

5. APPLYING THE VISUAL REQUIREMENTS SPECIFICATION LANGUAGE

5.1 Application on the Timeline Tool WCSS

The new specification language can be applied to the Timeline Tool WCSS as an example of what a full visual requirements specification might contain. In this chapter, we focus on a few key areas of the Timeline Tool's visual requirements. A specification for the entire Timeline Tool (excluding additional cluster information) is given in Appendix B. Portions of the specification within this chapter are segmented to allow commentary and to call attention to specific instances of visual requirements. The prose examples given here were all created using a simple word processor with capabilities for adding hyperlinks and images.

Using the visual requirements specification language, the specification document is able to uniquely and carefully follow the same design structure as the interface itself. An overview of the Timeline Tool design structure is provided in **Figure 5.1 and Figures 5.2 and 5.3.**

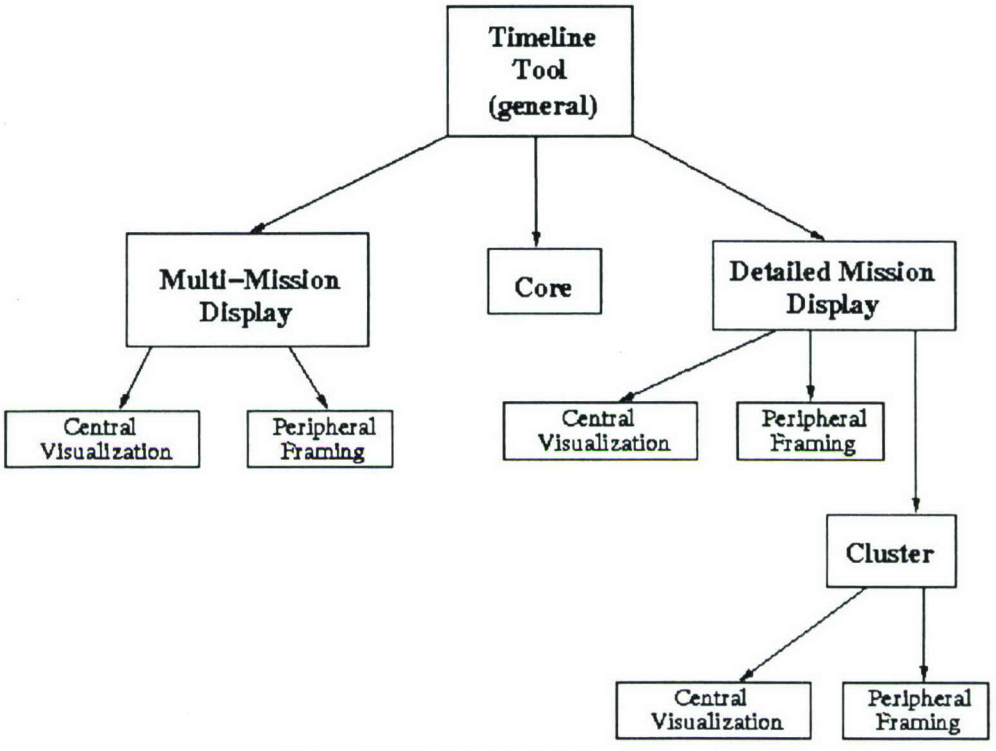


Figure 5.1 Timeline Tool Design Concept

The design concept for each area of the Timeline Tool follows a repeating pattern featuring a central visualization with additional attributes shown in the peripheral space. Following a design pattern allows work-orientation goals to be met at each sub-pattern level. It also cuts down on the variations required to specify the interface model. We begin specification with the multi-mission display and work inwards towards the lowest level of requirements. Using this top-down approach guards the specification from missing any subtle low-level details.

I. Multi-Mission Display Visual Specification
A)
Name: Multi-Mission Display
Link: Multi-Mission-Display
Attributes: None
Subsumed Parts: 2
AA)
Name: Mission Sorting Display
Link: Mission-Sorting-Display
Attributes: None
Subsumed Parts: 0

AB)
Name: Mission Selection and Core Display
Link: Mission-Selection-Core-Display
Attributes: None
Subsumed Parts: 2

ABA)
Name: Mission Selection Column
Link: Mission-Selection-Column
Attributes:
a) *Event:* Mission is selected (double click)
***Action:* Load corresponding detailed mission view for selected mission**
Subsumed Parts: Multiple

ABB)
Name: Core Displays
Link: Core-Displays
Attributes:
1) Timespan view = 24 hours
2) Time format = GMT
a) *Event:* Alert status change
***Action:* Change color of individual core display dependent upon alert status**
b) *Event:* Horizontal scrolling left or right
***Action:* Scroll forwards or backwards in time on all core displays**
c) *Event:* Double click on any part of an individual mission core
***Action:* Switch to Detailed Mission View of selected mission core**
Subsumed Parts: Multiple

ABBA)
Name: Core Display
Link: BA

ABBB)
Name: Time Indicator
Link: Time-Indicator
Attributes: None
Subsumed Parts: 0

Figure 5.2 Multi-Mission Display Visual Specification

The multi-mission display is specified into two separate divisions. The first is the relatively simple *Mission Sorting Display*, which is where missions are able to be filtered and sorted dependent upon user input criteria. The second and more complex is the *Mission Selection and Core Display*. Within the *Mission Selection Column* (part

ABA), the ability to select and bring up a mission in the detailed mission view is captured by dynamic attribute *a*. This part has multiple subsumed instances of individual mission buttons, but the fine details concerning those parts were omitted for brevity (see Appendix B for a complete listing). Part ABB describes the *Core Displays* area of the multi-mission view. This part is probably the most important item in this portion in terms of retaining a work-centered focus throughout development. As a result, many static and dynamic attributes are given to it in order to capture detail. Here we find a vital alert-related requirement in attribute *a*. This dynamic attribute covers the situational awareness support feature, allowing the user to monitor many missions and be notified when a mission goes on alert. If a requirement such as this were left unspecified, the alert notifications could be implemented incorrectly or non-optimally, possibly eliminating much of the work supporting ability of the multi-mission display. Note that all attributes specified at the *Core Displays* level will apply to each individual core display which is listed as a child part. The *Core Display* part given in ABBA contains only a link to part BA. As parts such as the core display are reused throughout the Timeline Tool, here we delineate the full specification to one defined location, part BA. Images for each part's link contents can be found alongside the full specification given in Appendix B.

II. Detailed Mission Display Visual Specification
B) Name: Detailed Mission Display
Link: Detailed-Mission-Display
Attributes: None
Subsumed Parts: 2

BA) Name: Flight Data Depiction
Link: Flight-Data-Depiction
Attributes: None
Subsumed Parts: 3

BAA) Name: Departure Data
Link: Departure-Data
Attributes: None

Subsumed Parts: 4

BAB) Name: Central Core Timeline
Link: Central-Core-Timeline
Attributes: None
Subsumed Parts: 1

BABB) Name: Central Timeline Window
Link: Central-Core-Timeline
Attributes:
a) *Event:* Content change
***Action:* CALL respective data sources for child parts**
Subsumed Parts: 2

BABBA)
Name: Time Points
Link: Time-Points
Attributes:
1) Location = Along solid and dashed lines
2) Base color = white
a) *Event:* Time point flagged
***Action:* Change color of time point from white to black**
Subsumed Parts: 4

BABBB)
Name: Timeframes
Attributes:
1) Display type = lines/bars
Subsumed Parts: 4

BABBBA)
Name: Flight Capability Timeframe
Link: Flight-Capability-Timeframe
Attributes:
1) Location = top of central timeline window
2) Line type = dotted
Subsumed Parts: 0

BABBBB)
Name: Flight as Planned Timeframe
Link: Flight-as-Planned-Timeframe
Attributes:
1) Location = center of central timeline window
2) Line type = solid black
3) End points = diamond time points
Subsumed Parts: 0

BABBBD)
Name: AR Window of Opportunity Timeframe
Link: AR-Window-of-Opportunity-Timeframe
Attributes:
1) Line type = colored bar

a) *Event:* **CALL** go/no-go reservation
Action: Bar color change
Subsumed Parts: 0

BABBC)
Name: Status Indicators
Link: Status-Indicators
Attributes:
1) Location = attached to time point where time difference exists
2) Size = difference between projected time and actual time
3) Color = red for negative difference, green for positive difference
4) Direction = right for negative difference, left for positive difference
a) *Event:* **CALL** content feed
Action: Modify static attributes to accurate values
Subsumed Parts: 0

BABBD)
Name: Status Bars
Link: Status-Bars
Attributes:
1) Location = attached to end of flight as planned timeframe
2) Size = difference between project time and actual time
3) Direction = right for negative difference, left for positive difference
a) *Event:* **CALL** content feed
Action: Modify static attributes to accurate values
Subsumed Parts: 0

BAC)
Name: Arrival Data
Link: Arrival-Data
Attributes: None
Subsumed Parts: 4

Figure 5.3 Detailed Mission Display Visual Specification

The detailed mission view of the Timeline Tool has a much wider variety of requirements that must be captured in order to support the user's work practices. Starting at the highest level, the detailed mission view is divided into the *Flight Data Depiction* (core display, part BA) and the *Cluster Display* (to be covered later in this section, part BB). The *Flight Data Depiction* has three subsumed parts: *Departure Data, Central Core Timeline,* and *Arrival Data.* The departure and arrival data segments are peripheral boxes which surround the central display. They both contain similar content, which is arrayed and positioned within their allotted rectangular

space. The *Central Core Timeline* is where the majority of the detailed mission requirements lie. Within the *Central Timeline Window*, a dynamic attribute captures the behavior of the many various data streams responsible for all active content. As the *Central Timeline Window* contains a large amount of data arriving from different sources, it is effective to specify at this level that each will behave as a result of an incoming data feed.

The main two elements within the *Central Timeline Window* are the individual *Time Points* and *Timeframes* parts. *Time Points*, as might be described within an external toolkit vocabulary, are way-points throughout a mission which signify events. Each time point represents an event occurrence of note during a mission. As a way-point is achieved mid-mission, it is correspondingly "flagged" by the commanding officer. As a result, here there exist attributes describing the color and behavior of flagged/non-flagged time points. Time point location is also an important attribute as it ensures that each is placed along the overall mission timeline and not freestanding or floating. Eliminating clutter and capitalizing on the horizontal time axis layout allow *Time Points* to blend seamlessly into the workstation display.

Timeframes represent periods of availability and prediction estimation for flight-related events and activities. They are created as lines or bars in contrast to the points or icons used for mission way-points. Three examples of specific child timeframes are specified in order to show the variability for different child types. Note that for each type, there is a uniquely assigned line type (dotted, solid, and colored). This line type, along with location, distinguishes each timeframe from the next, again allowing a user to easily identify and view multiple data streams in a coherent, non-chaotic way.

The final two parts of the *Central Timeline Window* are *Status Bars* and *Status Indicators* respectively. These two elements work in a similar manner to display variances from originally intended flight times and schedules. Together, they identify when a mission is ahead of or behind schedule by their color, size, and direction. *Status Bars* attach directly to *Timeframes*, while *Status Indicators* connect to *Time Points*. Although there is no direct method for linking this pair of related parts, static attributes are applied referencing correlated part location. By specifying clearly the details of what and where these parts should be, the work-oriented visual aiding possesses continuity over the entire user interface. A user no longer must "eyeball" how far behind a mission is, nor must they compute differences in current time and estimated time of arrival. Instead, automated agents compute and stream data to the interface which displays *Status Bars* and *Status Indicators* for the user to visually inspect. The turnaround time in reaction to these visual aids is nearly simultaneous due to their favorable location and color-coding. Diagrams and sketches describing *Status Points* and *Status Indicators* are included in Appendix B as link reference materials.

Due to the large volume of requirements and parts associated with the detailed mission view, many parts and attributes have been omitted in an effort to emphasize those which best display various visual requirements categories. Any discrepancies in numbering or labeling are as a result of these omissions. The full detailed mission view specification can be found within Appendix B.

The final portion of the Timeline Tool and the second half of the detailed mission view is the *Cluster Display*. Its specification can be found in Figure 5.4.

BB)
Name: Cluster Display
Attributes: None
Subsumed Parts: 6

BBA)
Name: Diplomatic Permissions Cluster
Link: Diplomatic-Permissions-Cluster
Attributes: None
Subsumed Parts: 3

BBAA)
Name: DIP End Tab
Attributes: None
Subsumed Parts: 2

BBAB)
Name: DIP Time Window
Attributes:
a) *Event:* Disagreement of DIP/nation clearance
Action: Change status colors to reflect disagreement
Subsumed Parts: 2

BBABA)
Name: Nation Overflight Indicators
Attributes:
1) End Points = Diamond Time Points
2) Indicator Type = Horizontal Color Bar
a) *Event:* Multiple Nation Overflights
Action: Different color shading for each overflight
Subsumed Parts: 0

BBABB)
Name: DIP Clearance Indicators
Attributes:
1) Indicator Type = Horizontal Color Bar
a) *Event:* Multiple DIP Clearances Present
Action: Different color shading for each indicator
Subsumed Parts: 0

BBAC)
Name: DIP End Tab
Link: BBAA

BBD)
Name: Airfield Cluster
Link: Airfield-Cluster
Attributes: None
Subsumed Parts: 3

BBDA)
Name: Airfield End Tab

Link: Airfield-End-Tab
Attributes:
1) Height = 100 pixels
2) Width = 80 pixels
Subsumed Parts: 5

BBDB)
Name: Central Airfield Timeline
Link: Central-Airfield-Timeline
Attributes:
1) Height = 100 pixels
2) Width = 520 pixels
a) *Event:* Mouse over any subsumed part
Action: Display Time Parameters Tooltip
b) *Event:* Alert/Violation
Action: Change Color-coding
c) *Event:* Port Display Region
Action: **CALL** get projected arrival times for airfield
Subsumed Parts: Multiple

BBDBA)
Name: Individual Timelines
Link: Individual-Timelines
Attributes:
a) *Event:* Content Data Updates
Action: **CALL** respective data sources for subsumed parts
Subsumed Parts: 5

BBDC)
Name: Airfield End Tab
Link: BBDA

BC)
Name: Time Indicator
Link: ABBB

Figure 5.4 Cluster Display Visual Specification

Of the six clusters developed for the completed Timeline Tool, the *Diplomatic Permissions* and *Airfield Clusters* are specified here. Both clusters follow the same design pattern set forth in the Timeline Tool design concept. The *Diplomatic Permissions* or *DIP Cluster* contains a pair of bounding boxes and a central visualization depicting nation and permissions information. The *DIP Time Window* (part BBAB) outlines one attribute which defines behavior when gaps exist between permissions. This is critical to the mission planner, as not obtaining appropriate

permissions prior to entering foreign airspace can be a high priority alert and dangerous risk. The *Nation Overflight Indicators* and *DIP Clearance Indicators* both follow the same formatting as that of *Timeframes* within the core display. Using color shading makes it easy to view when clearances and nations overlap.

The *Airfield Cluster* varies from the other clusters only slightly in its content and behavior. Dynamic attribute *c* within the *Central Airfield Timeline* captures the dynamic requirement of displaying visual information only when a specific port is in range. As this behavior is defined at the individual cluster level, it will only apply to the *Airfield Cluster*. As the *Airfield Cluster* itself contains five separate data streams, each is divided into an individual timeline which then draws in and displays data. As a result, the *Airfield Cluster* performs exactly as it was designed; providing associative mission data in a location and formatting which supplement but do not distract from the core mission summary.

In an effort to provide a well-rounded understanding of the entire Timeline Tool package, Figures 5.5 and 5.6 display several screen shots taken from initial prototypes of the Timeline Tool system.

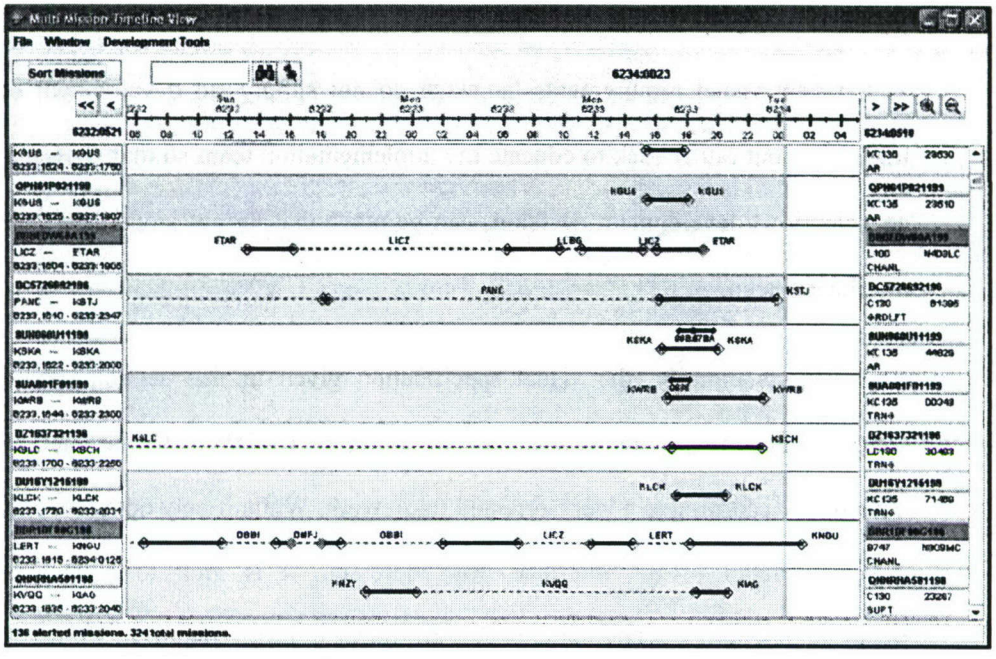


Figure 5.5 Multi-Mission Prototype

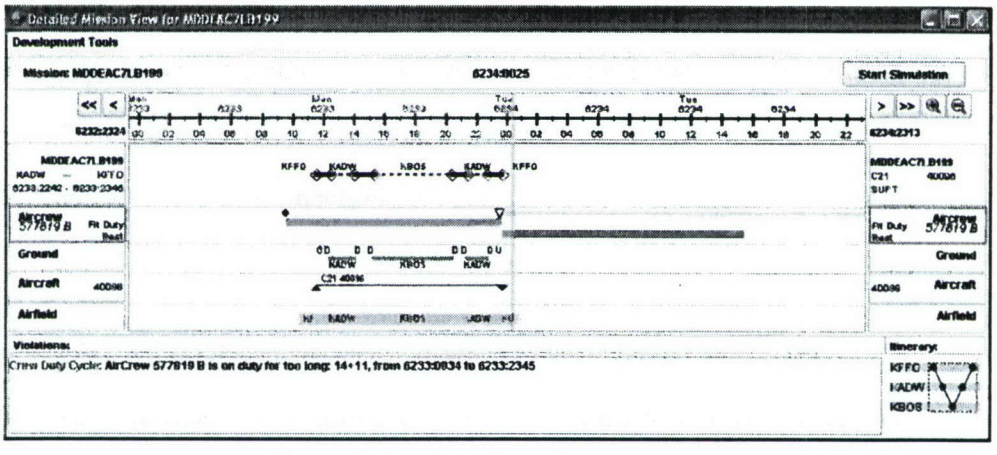


Figure 5.6 Detailed Mission View Prototype

Generally, the prototypes of both the multi-mission display and the detailed mission display are exact depictions of the design screens. However, several subtle differences have been incorporated. Within the multi-mission view, the ability to search and sort missions has been integrated into a smaller area, as well as into the program bar at the top of the screen. Also, the mission selection buttons on the left hand side have been dropped in favor of using the mission tabs themselves as detailed mission view links. Changes such as these reflect necessary and obligatory design

trade-off points that occur while software is being developed. Specification tools such as the new visual requirements language do not simply rid development of these trade-offs, but rather seek to educate the implementation team so that wise and work-conscientious development decisions can be made that do not jeopardize the overall interface design.

In its entirety, the visual specification given in this section provides the concrete visual requirements necessary for this portion of the software interface to be developed according to a work-oriented framework. While it may not seem as if every mundane detail of the interface was addressed, it is vital to recall that only requirements and interface parts which are intrinsic to the nature of the work being done need be specified. It is this collection of UI elements which will constitute a working WCSS when realized and developed according to the design framework. This realization enables the designer to possess a certain amount of flexibility in specifying a design, as well as keeping the development of a WCSS from being too impractically stringent in its requirements.

The visual requirements specification language captures the valuable display information and provides a channel for communicating it effectively to the developer, ensuring a functional work-centered software system as a result. In contrast, using other current approaches (with only UML) leave behind valuable visual design requirements. These details, such as the assimilation of many data layers into the simple timeline view, comprise the work-centered software representation which allows the completed functional application to meet the user's need in an optimal manner. Without specification of such details, it is unlikely that any work-centered software system can be accurately realized in a non-specialized development setting. The visual requirements specification language contributes significantly towards the

end goal of work-centered software being developed through a coherent software process.

6. INTEGRATING THE VISUAL REQUIREMENTS SPECIFICATION LANGUAGE

6.1 UML Augmentation

In order to be applicable to standard software engineering practices, the new specification language must integrate well with existing software modeling methodologies. This is quite easily done via the UML's various facets for extension. In Figures 6.1 and 6.2 are two examples of integration: using the UML package notation to group functional (UML) specification and related visual requirements specification together, and using the UML comment notation to include links to visual specification at appropriate design points. A newly devised visual specification language complements the UML with addition capability for capturing overall system design, both process and presentation.

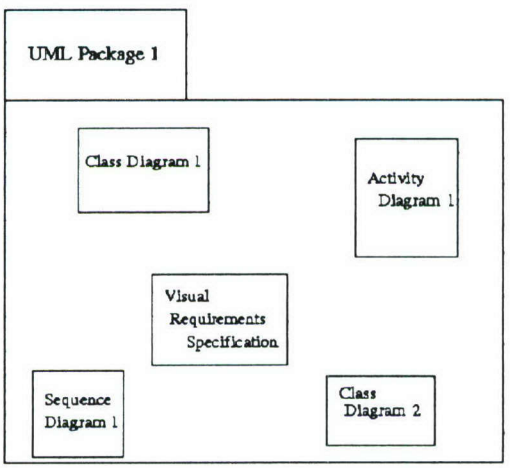


Figure 6.1 UML and Visual Requirements Language Package Integration

The UML package diagram is excellent for integrating additional specification materials because of its grouping construct which allows the combination of elements into higher-level units [2]. Packages are quite useful for large-scale and complex systems as they give a way to appropriately layer design components into compartmentalized groupings. In the example given in Figure 6.1, a generic UML package is shown with various other UML diagrams contained within it. In this mode of application, a visual requirements specification would be inserted when it is directly related to a majority of the diagrams within the overall package. For example, the class diagrams shown here might be specifically related to certain visual elements and display pieces, such as buttons or controls. A sequence diagram could show background computations and interactions done as automation support which is then linked to specialized UI display panels. An activity diagram might represent functional components which undergo many transformations during usage and as a result effect the display transformations as well. These few examples are just a sampling of ways in which the most popular of the UML's diagrams would be usable alongside visual specification. The usage of the package diagram is directly helpful when the back-end code is directly related to the user interface and work-aiding display. Interrelated classes and extensive communications between interface and functional code are simplified through the usage of packages.

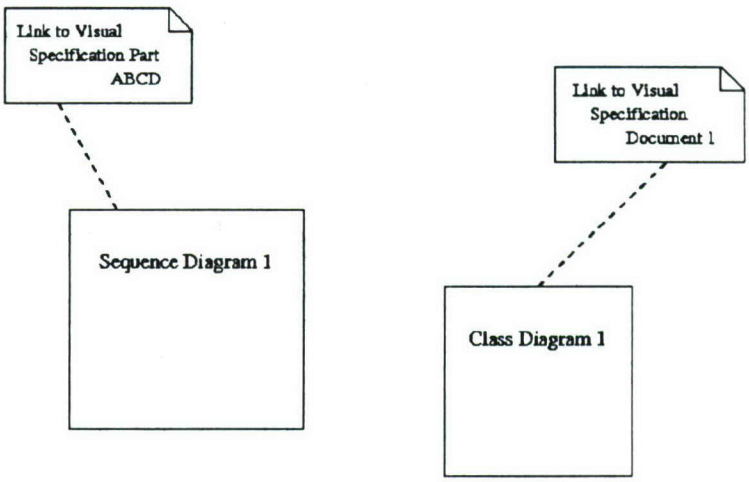


Figure 6.2 UML and Visual Requirements Language Comment Integration

The UML comment notation is much more flexible in terms of usage than that of the package diagram. UML comments can be inserted into any UML diagram via a dotted line and corresponding comment box. In this manner, any UML diagram can receive additional preface and explanation. Consequently, this notation is perfect for the insertion of visual requirements specification material. Since visual requirements can be linked not only to classes, but also sequences, activities, components, or use cases, it is advantageous to be able to insert linkages to visual specification wherever necessary to do so. This goal is achieved by utilizing the comment notation to provide necessary connections to specific visual specification parts. The chaining together of functional UML diagrams with visual specification provides the developer with the means to understand how the functional code is related to the visual display panel and resulting work-context. It also eases the difficulty of integrating the final compiled code with the visual interface by enabling development of both to proceed together. Using UML comments to fuse visual specification together with functional design models prepares the developer to create accurate high-quality work-centered software which meets both its visual and non-visual requirements.

6.2 Work-Centered Software Process

A critical question for the future of work-centered software is how it can be effectively scaled to a large scale development project. As a prototype design technology, Work-Centered Design theory has yet to be tested and employed on a system involving a large development team. In each software system developed using WCSS theory, the overall design and development team has been composed of a relatively small amount of cognitive and software specialists. Therefore, it is of worth to consider how work-centered practices might be applied to a development environment of more than fifty people. For work-centered software to be reasonably developed in the broader software community, its principles must be incorporated into a standard software process. Each phase of such a software process must blend the aspects of work-centered theory with those of traditional practice, producing a coherent lifecycle model. Creating such a linkage within the software development chain is complicated enough, despite attempting to couple it with work-centered practices. Thanks to standardized tools, such as the UML, connecting the various development phases of prevailing software development has been made manageable. Unfortunately, little to no tools and resources have been developed to aid the integration of WCSS principles into a software process. The success of future WCSS deployment hinges upon the creation of software design tools which can capture and communicate the essentials of this emerging design concept.

As mentioned in section 2.1.2, some research has already been devoted to the area of Work-Centered Design. This starting point can be viewed as the foundation for a work-centered software process. By involving both cognitive scientists, software engineers, and other domain experts, the initial framework for a work-centered

software system is in place. However, it is after this point that the amount of resources, tools, and methods becomes strikingly sparse. It is not realistic to propose that a work-centered focus can be maintained throughout development without the assistance of a specific set of direct communication methods. In the effort to begin to amend these deficiencies, the visual requirements specification language aims to be a contributing software development tool for maintaining a coherent work-centered software development lifecycle. However, the language itself along with its UML augmentation does not come close to filling the entire need for work-centered development tools. Figures 6.3 and 6.4 illustrate two different views of the stages necessary to develop work-centered software.

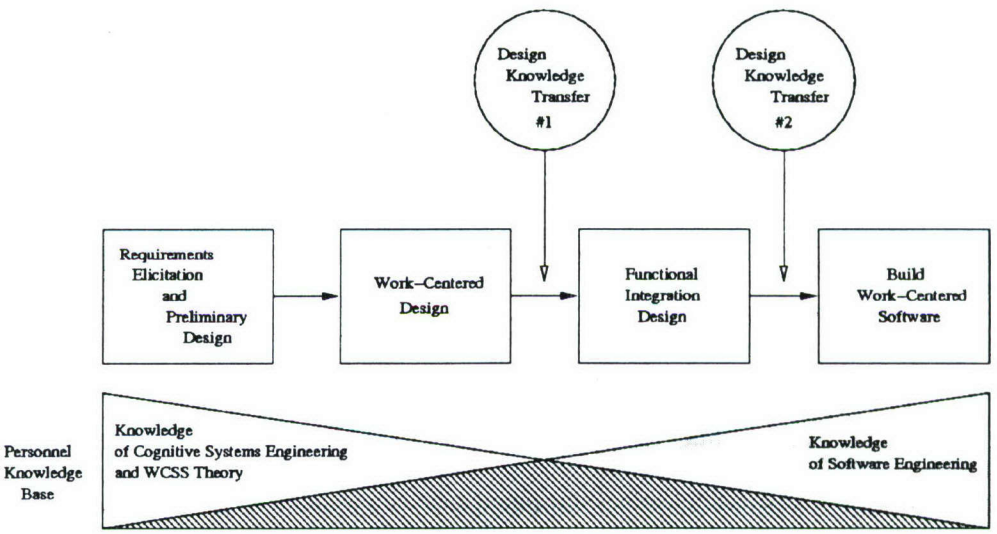


Figure 6.3 WCSS Development Process

Figure 6.3 displays a general overview of standard work-centered software development. It begins with the elicitation of requirements from the problem space, and the construction of preliminary design ideas which will solve the problem. In the case of a work-centered support system, this stage is completed mainly by cognitive experts who are able to draw out the work-context and establish a knowledge capture of the worker's perspective. From this knowledge capture data, a complete work-

centered design is created which constitutes the foundation for a work-centered support system. The design itself intrinsically supports the work necessary to satisfy the problem requirements. Once the work-centered design is complete, it is handed off to a developer for the remaining stages.

The developer has the job of supplementing the work-centered design with components and details of a functional system design. This portion of the design is composed of items such as data structures, platforms, data types, and algorithms. While the work-centered design specifies all the necessary items for a work-aiding system, it does not contain vital implementation planning and design. As a result, this transition is labeled as design knowledge transfer #1. The work-centered design is exposed to risk as it is transferred into the hands of a developer to complete. Once the full design is complete (both work-centered and standard functional designs), coding and development can begin.

The final stage is to have the software built by a development team. The amount of personnel involved in this stage is likely to be a much larger number than that of earlier stages. This creates the second design knowledge transfer, as well as many associated risks. Here, the design must be correctly interpreted and implemented accurately, most likely by programmers who have the least amount of knowledge about the work-context. Decisions made in this stage could dramatically effect how successful the resulting WCSS turns out to be. It is of utmost importance that the transition to this stage be done fluidly and comprehensibly, otherwise confusion may lead to a disappointing end result.

At the bottom of Figure 6.3 is a chart describing the knowledge of personnel working on the project throughout each development stage. This is done to show how

the further a project progresses, the more it moves away from a cognitive and human factors focus and towards a software engineering center. It further illustrates the necessity for coherency throughout the work-centered lifecycle in order to maintain the core facts and requirements which judge the final software as effective or ineffective.

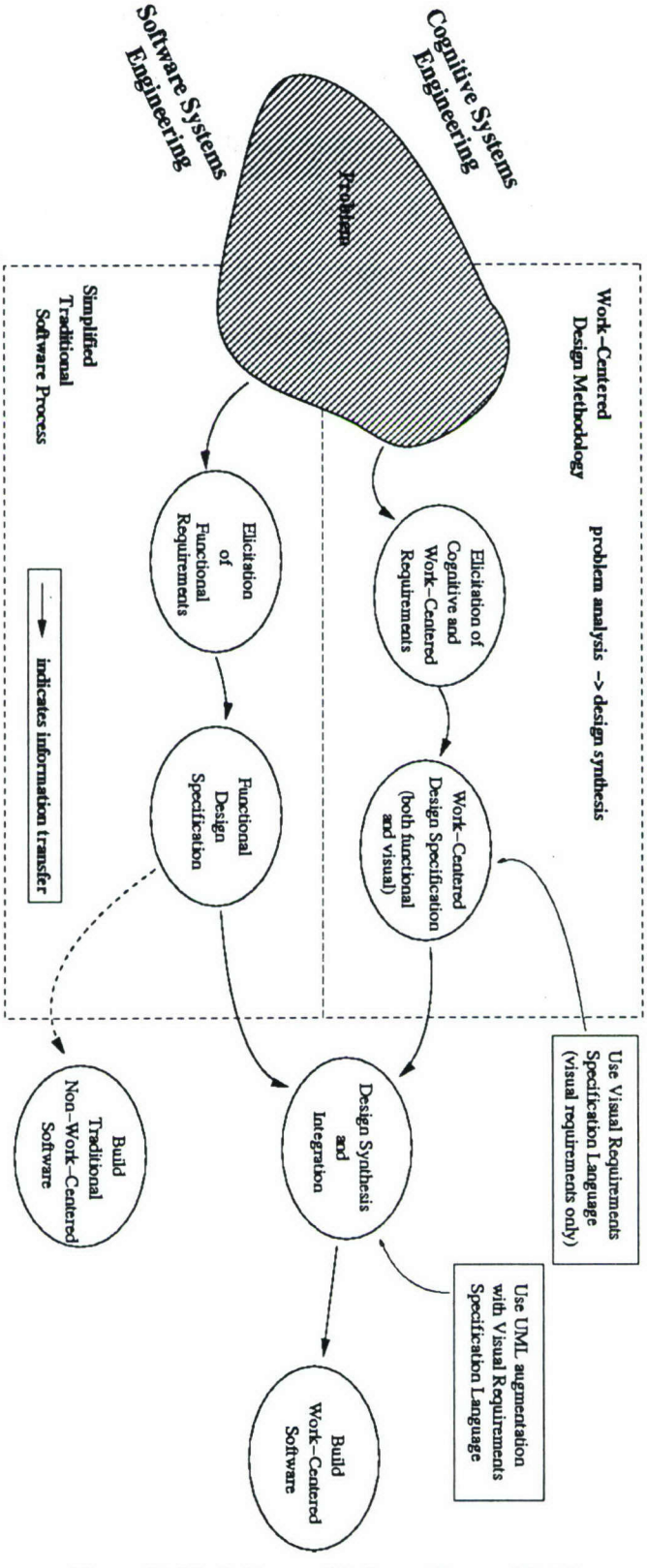


Figure 6.4 Work-Centered Software Process Model

Figure 6.4 shows a more complete sketch of a work-centered software process. While it is somewhat abridged, leaving out aspects such as testing, maintenance, and lifecycle type (such as waterfall or spiral), it displays conceptually the various essential components of a plausible work-centered software process. The two rectangular dotted boxes separate the key aspects of each engineering discipline, showing how the two differ and eventually must relate. The bottom half of the figure can be seen as what constitutes a traditional, non-work-centered approach. The top half represents the work-centered design framework. Indications are made where the visual requirements specification language and its UML augmentations can be employed throughout the process. The presented process within Figure 6.4 draws out several noteworthy characteristics and implications.

The first key characteristic is that although both software engineers and cognitive scientists share the same problem space for developing a software solution, they utilize that space in alternative ways. This diversion of both requirements and design is caused by a separation in design intent and perspective. The cognitive perspective looks for the overall work-context and framework in which the problem is set. It encapsulates this framework by constructing an ontology made up of key artifacts which are related to the user's work. It is from this distinction that a work-centered design is articulated and devised. From the software perspective, the problem space is a logical and mathematical dilemma which can be effectively managed and computed using a series of data structures and calculations. The functional design is to be a robust and efficient solution to the problem's requirements. Each discipline approaches the problem from a different perspective, achieving two different results which both contribute to the final software.

The second critical characteristic of a work-centered software process is the notion of coherency. It is not enough to simply state that the cognitive and software disciplines are different and must both be employed to complete a project successfully. Coherency must be an attribute which is incorporated into both schools of development. Both sets of design teams and models come together in the design synthesis and integration stage. How well concepts and ideas have been communicated becomes most apparent only once the final software has been delivered. While the majority of this paper speaks of developing tools and methods which enable cognitive and human factors experts to associate and create specification for software engineers, coherency addresses not only the cognitive science field, but also the software field. Software engineers must adhere to standards which permit and encourage communication during the final steps of implementation to avoid the destruction of vital work-centered details. Otherwise, the framework for coherency will be lost in the penultimate stages of the project. Coherent communication between diverse design and development teams can ensure a successful software venture.

Finally, work-centered design and theory are still relatively new technologies. It is without a doubt that they have not been completely exhausted in terms of design components, stages, and theories. As the work-centered paradigm continues to evolve, a more distinctive picture of a work-centered software process will be clear. Details and information regarding additional portions of work-centered theory and its relation to software engineering are addressed in Chapter 7.

7. REVIEW AND RELATED SUBJECTS

7.1 Review of the Visual Requirements Specification Language

As a software specification tool, the visual requirements specification language based upon the user interface markup language is a positive first step towards a unified work-centered design model. The visual specification language contributes considerably in the area of bridging software engineering with cognitive engineering and human factors design by allowing work-centered visual designs to be specified semi-formally into a UML object model. By employing the language on a visual interface, each portion and part of the display can be decomposed and identified at an appropriate level of detail. Using an XML-compliant scalable structure makes the language flexible enough to capture visual requirements of any user interface design, yet simple enough that it can be employed by cognitive scientists and human factors experts. By encapsulating the crucial visual requirements inherent to a work-aiding interface display, the risks associated with transmission of vital work-centered design artifacts are significantly reduced throughout development.

The visual requirements specification language employs four categories of tags to capture visual display content. Each attribute tag encompasses a specific area of work-related content essential to the creation of a work-aiding display panel. The language separates interface objects into distinct parts via unique labeling and

associated class names. Its nesting structure permits parent/child relationships and easy inheritance of specific design attributes. The language's static and dynamic attribute fields pertain to any and all characteristics of individual UI parts. Behavior of individual and collective interface parts is distinctly and accurately modeled using action/event pairs within dynamic attributes. An additional link module allows designers to connect useful multimedia and documentation directly to effected parts. The entire language follows a simple template layout, making the creation of specification documents easy using any text editor.

In addition to its definitive structure and set of features, the visual requirements specification language can be integrated into a standard UML object model using two different methods. The UML package structure allows visual specification to be included along with other visual-related functional diagrams such as classes, sequences, and activities. The UML comment notation gives easy access to individual UML diagrams which can be linked to specific visual interface parts. By connecting the visual requirements with the overall design model at large, developers are less likely to omit important work-centered details. Having all of a specification in a single document eliminates excessive materials and reduces the amount of stress on a developer to integrate miscellaneous system components.

Creating coherent, stable work-centered software using the visual requirements language is less prone to detrimental setbacks and misinterpretation mistakes typically present when developing work-centered software. The visual requirements specification language serves as a bridge on which software engineers can receive and comprehend cognitive design strategies which are becoming more and more prevalent in mainstream system development. Using the visual requirements

language ensures the transmission of vital work-centered visual details from design to development.

7.2 Related Subjects

7.2.1 Inclusion of Cognitive and Work-Context Data

During the process of work-centered design, large amounts of work-context data is collected during elicitation and knowledge capture stages. This information is collected by a team of cognitive experts studying the field of practice in which the WCSS will reside. Large quantities of data regarding the entire work operations context make it easier to analyze and create a work-centered design which is an exact match to the needs of its users. Once a design has been established, the cognitive data serves as justification for the various design decisions and aspects of aiding characteristic of a WCSS.

Current practices indicate that an enormous amount of this cognitive knowledge capture material should be transferred to the system developer. The justification for this practice being that the more information a developer obtains, the more easily they will understand what the work-centered design is attempting to achieve. However, in attempting to educate the developer on work-centered practices, the design team inadvertently encumbers the developers with the excessive cognitive knowledge base. There are several reasons why this encumbrance occurs and why the thinking behind it is flawed.

First, the developer and development team cannot possibly utilize all of the data present in the transcripts of work-orientation data. These documents often number in the hundreds of pages, making it a burdensome task to attempt to relate all

of its information to the final design presented for coding. Second, the development team is not a set of cognitive and human factors specialists. Their projected ability to reasonably and accurately understand and apply the information contained in knowledge capture reports is unlikely at best. Sending large quantities of this data raises the potential for wrong interpretation even more. Finally, the coding team typically has enough tasks and assignments without the addition of comprehending supplementary cognitive material. Software projects continue to struggle with deadlines and other setbacks due to external reasons, gross documents on work practices and environment need not add to this collection of burdens.

However, the original question answered by the cognitive experts in delivering the work domain capture remains. How much information about the work context should be included in a design transfer? Is such a design safe without the inclusion of any? Different views are held on both sides of this issue. This paper does not state an absolute amount necessary; rather it supports the view that all essential cognitive and work-related materials can be represented through the use of software development tools and methods. By creating unique and specific methods to capture work-related information, there need not be an additional burden in work-centered software development. Since information regarding work-centered requirements must be transmitted at some particular point during the software process, it can and should be linked directly with any cognitive materials necessary to fully comprehend and make well-educated design decisions at trade-off points.

The ineffectiveness of a complete work knowledge capture transfer has already been identified by the WCSS and Work-Centered Design experts at AFRL. In light of this, research in the area of providing work-design representations in a semi-formal syntax is already underway. This encapsulation of cognitive context data is

similar in nature to materials which would be found in the visual requirements specification language's toolkit vocabulary used to provide additional informative data for visual specification parts. It is plausible to say that the new specification language could be linked and integrated with a cognitive design specification language or structure in future development research.

7.2.2 Functional Work-Centered Requirements

This paper specifically addresses the problem of specifying and communicating work-centered visual requirements. However, throughout the research done on WCSS theory, it is quite apparent that there exist more than simply visual requirements associated with standard WCSSs. Although it is difficult to fully illustrate the variety of WCSS requirements because WCSS theory continues to evolve and grow, it can be estimated that there are several other types of requirements aside from visual within typical work-centered designs. Locating and identifying all the various types of requirements is a task which will be of utmost importance as work-centered software grows in popularity and corporate acceptance. Once classified, methods for procurement and transmission can be made to further serve the creation and acceptance of a widespread work-centered software process.

One particular area of WCSS requirements noted during research was that of automation. Automation agents serve as the functional half of a WCSS. These automating devices directly aid the user by performing calculations in the background to simplify the user's job and provide additional work support. Obviously these agents do not have visual requirements as they rarely, if ever, are seen on a display screen. Instead, these agents can be categorized more accurately as standard functional requirements such as those captured by the UML. In essence, automation

requirements can be viewed as additional functional calculations to be performed by the central computing portion of the software. While these agents may never be devised by a standard software engineer, since they would be seen as unessential extra computations and objects, they are nonetheless helpfully supportive to the user and the accomplishment of work. Whether all of these types of requirements can be fully addressed and captured using the UML has yet to be determined. However, it can be assumed that at least a portion of automation requirements would fall into this category. Methods and techniques for communicating and integrating these requirements into a final design is also yet to be addressed, but will be integral in the further understanding and completion of work-centered software at a higher level.

7.2.3 Integrating the Visual Requirements Specification Language into a Development Environment

As a language based upon an XML-compliant meta language (UIML), the visual requirements specification language still retains its ties to the parsable and syntactical structure of the XML. Despite modifications made to make the language more suitable to cognitive and human factors personnel, the overall document structure, although rearranged, remains intact. There exist many possibilities for this attribute, namely the ability to integrate the language into a higher-level development environment. Programs which are capable of rendering prose contents into an immediate visual depiction could be employed for the creation and review of design specification. Moving the opposite direction, applications could be made which allow a designer to first draft a screen design using drawing and palette tools, then specify that design using built-in click and drag capabilities, and finally view an automatically generated text specification! As web design programs make use of such features for displaying previews and code within the same viewing area, so too can specification

languages such as the visual requirements language be employed to make the job of designing and specifying a more computer-aided one. Further abilities in transferring and editing designs across long distances more manageably can be researched and developed as a result.

7.2.4 Expanding the Visual Requirements Specification Language

As mentioned earlier in section 4.2.7, the UIML basis for the visual requirements languages contains many other specialties and constructs than just those which were integrated into a specification language. As an XML-language, the visual requirements specification retains all the properties thereof, meaning that changes and modifications can be made to upgrade the language to current standards and practices. As research continues to be done in WCSS design theory, specification needs will become more prevalent and apparent in particular requirements areas. As these needs are illustrated, specification resources can be re-evaluated for usefulness and accuracy. Modifying the visual requirements language to include more UIML related materials, or additional un-related cognitive specification is available via the open XML document definition standards.

APPENDIX A

INSTRUCTIONS FOR EMPLOYING THE VISUAL REQUIREMENTS SPECIFICATION LANGUAGE

This appendix outlines instructions for applying the visual requirements specification to specify a work-centered interface display for a development team.

1. Review the interface design and determine where logical divisions can appropriately be made.
    (a) These divisions should be natural in separation. For example, two distinct screens would merit two distinct portions of specification.
    (b) Initial divisions will typically be broad (i.e. an entire screen, a large interface panel, etc) so that appropriate sub-divisions and specification can subsequently be made.
2. Divide the interface into separate logical and physical portions which partition the entire interface at the topmost level of abstraction.
    (a) Each of these portions will be the "parent" of all parts contained within them.
3. Select one of the top-level portions to specify.
4. Give this portion of specification a unique identifying header (A, AB, etc.) and name (Graph Display, Spreadsheet Panel, etc.).
5. Identify any static (unchanging) attributes (colors, shapes, text) for this part and all subsumed parts.

(a) Remember that any static attributes specified at the parent level will apply (where applicable) to subsumed child parts as well (e.g. use all red text).

6. Specify a static attribute under the "Attributes" category, by giving each a unique header (1, 2, 3) and stating the property in an X = Y format.
7. Be sure to use appropriate indentation to separate attributes from other specification.
8. Repeat step 6 for any and all remaining static attributes for this part.
9. Identify any dynamic (changing) attributes (motion, actions, behaviors) for this part and all subsumed parts.
    (a) Remember that any dynamic attributes specified at the parent level will apply (where applicable) to subsumed child parts as well (e.g. tool tip text)
10. Specify a dynamic attribute under the "Attributes" category, by giving each a unique header (a, b, c).
11. List the event which activates the dynamic attribute next to an "Event" header.
12. List the action which is taken as a result of the corresponding event beside an "Action" header.
    (a) All dynamic attributes must follow this event/action pair formatting.
13. Appropriately apply the CALL keyword where necessary within the event/action pairs used for dynamic attributes.
    (a) The CALL keyword references external functional logic and computations done outside the interface display.
14. Repeat steps 9 - 13 for any remaining dynamic attributes.
15. Be sure to use appropriate indentation for dynamic attributes to separate them

from other specification.

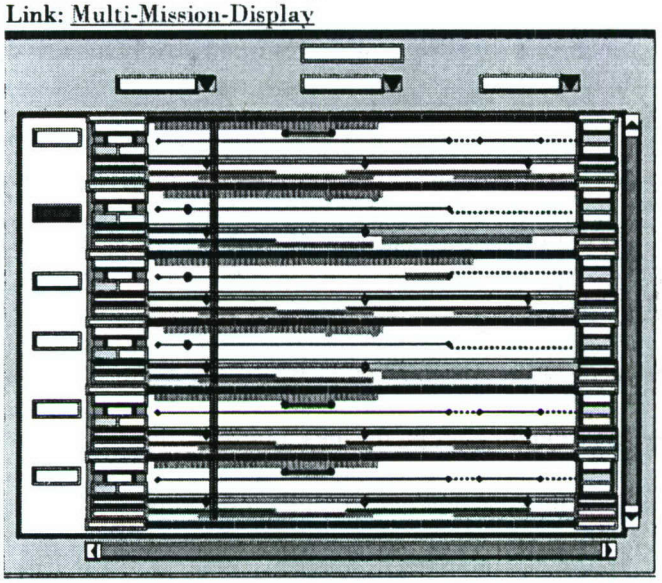
16. Fill out an entry for this part in the external toolkit vocabulary, referencing it using its part name.
    (a) Here additional details regarding cognitive and work-context information can be stated. For example, how a specific part of the interface mimics and represents portions of the work-context can be detailed in the vocabulary entry.
17. Locate any additional media and materials related to this part or its attributes.
18. Use the Link category to create linkage between additional materials and prose specification.
    (a) Links can be images, video, other documents, or other media.
19. Repeat steps 4 - 18 for each subsumed part within this part under the "Subsumed Part" heading.
20. Use indentation to visually signify parts which are children of others. Make sure to follow the same formatting for the entire specification.
21. Repeat steps 3 - 20 for each top-level portion of the interface.
22. Combine all specification materials (syntax, external vocabulary, and link media) into a single document for delivery to developer.

APPENDIX B

TIMELINE TOOL WCSS VISUAL SPECIFICATION

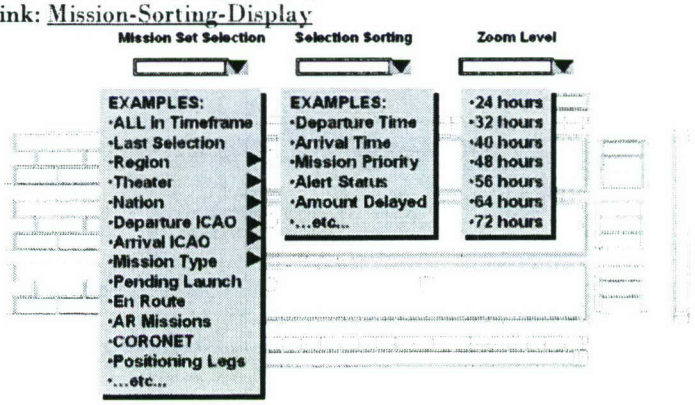
Timeline Tool Wide Spiral 1
AFRL/HECS
I) Multi-Mission Display

A)
Name: Multi-Mission Display
Link: Multi-Mission-Display



Attributes: None
Subsumed Parts: 2

AA)
Name: Mission Sorting Display
Link: Mission-Sorting-Display



Attributes:

1) Location = top of screen

a) *Event:* Selection of filtering criteria

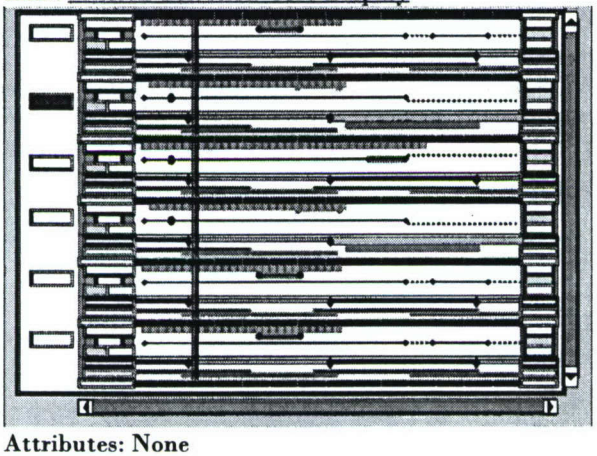
Action: Display appropriate missions/views

Subsumed Parts: 0

AB)

Name: Mission Selection and Core Display

Link: Mission-Selection-Core-Display



Attributes: None

Subsumed Parts: 2

ABA)

Name: Mission Selection Column

Attributes:

a) *Event:* Mission is selected (double click)

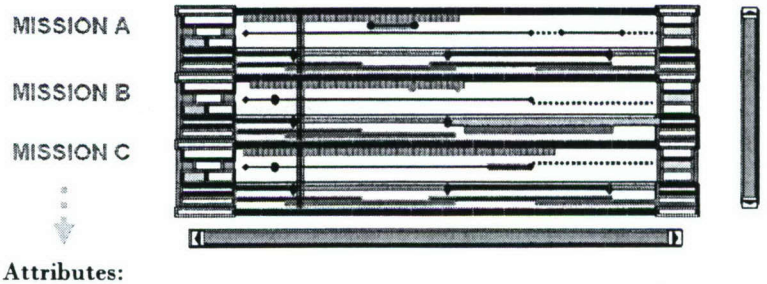
Action: Load corresponding detailed mission view for selected mission

Subsumed Parts: Multiple

ABB)

Name: Core Displays

Link: Core-Displays



Attributes:

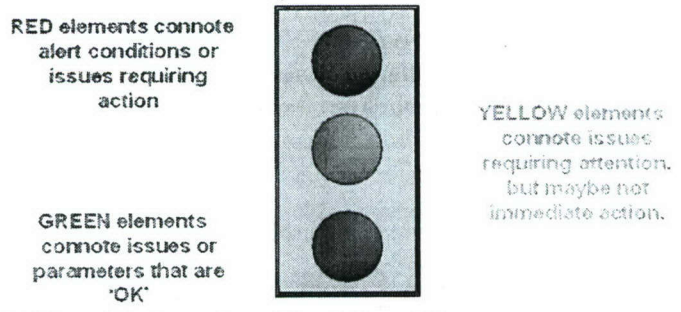
1) Timespan view = 24 hours

2) Time format = GMT

a) *Event:* Alert status change

Action: Change color of individual core display dependent upon alert status

Link: Alert Status



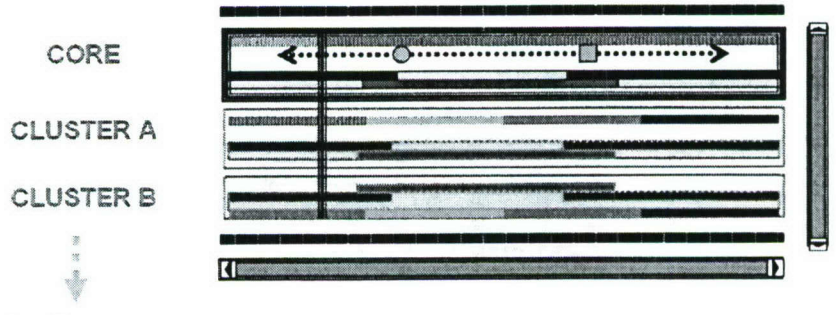
b) ***Event:*** **Horizontal scrolling left or right**
Action: **Scroll forwards or backwards in time on all core displays**
c) ***Event:*** **Double click on any part of an individual mission core**
Action: **Switch to Detailed Mission View of selected mission core**
Subsumed Parts: Multiple

ABBA)
Name: Core Display
Link: BA

ABBB)
Name: Time Indicator
Attributes:
a) ***Event*****: Time Position Movement**
Action**: CALL current time position**
Subsumed Parts: 0

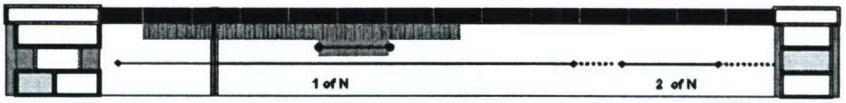
Section II) Detailed Mission Display

B)
Name: Detailed Mission Display
Link: Detailed-Mission-Display



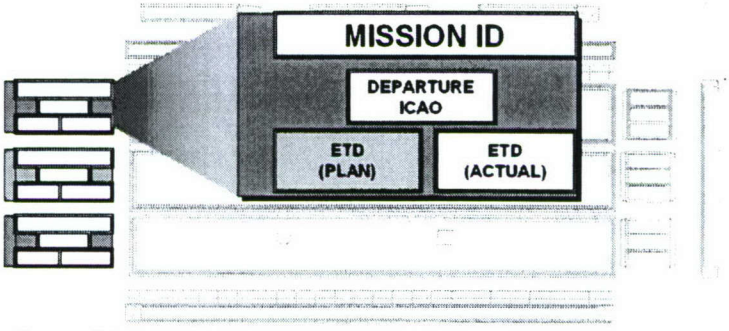
Attributes:
a) ***Event*****: Alert Status Change**
Action**: Change Mission Color Scheme, green for "clear", yellow for "caution", red for "warning"**
Link: Alert Status
Subsumed Parts: 2

BA)
Name: Flight Data Depiction
Link: Flight-Data-Depiction



Attributes: None
Subsumed Parts: 3

BAA)
Name: Departure Data
Link: Departure-Data



Attributes: None
Subsumed Parts: 4

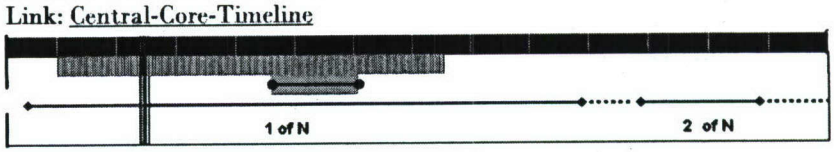
BAAA)
Name: Mission ID
Link: Departure-Data
Attributes:
1) Location = Top of Departure Data (BAA)
Subsumed Parts: 0

BAAB)
Name: Departure ICAO
Link: Departure-Data
Attributes:
1) Location = Center of Departure Data (BAAB)
Subsumed Parts: 0

BAAC)
Name: ETD (PLAN)
Link: Departure-Data
Attributes:
1) Color = Shaded different than other parts within Departure Data (BAAB)
2) Location = lower left of Departure Data (BAAB)
Subsumed Parts: 0

BAAD)
Name: ETD (ACTUAL)
Link: Departure-Data
Attributes:
1) Location = lower right of Departure Data (BAAAB)
Subsumed Parts: 0

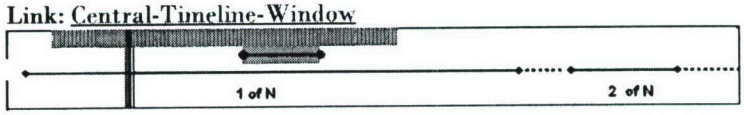
BAB)
Name: Central Core Timeline
Link: Central-Core-Timeline



Attributes: None
Subsumed Parts: 2

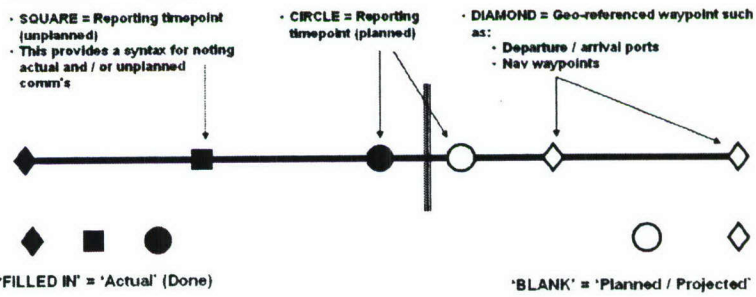
BABA)
Name: Time Index Bar
Attributes:
1) Content = GMT divisions
Subsumed Parts: 0

BABB)
Name: Central Timeline Window
Link: Central-Timeline-Window



Attributes:
a) *Event:* Content change
Action: **CALL** respective data sources for child parts
Subsumed Parts: 2

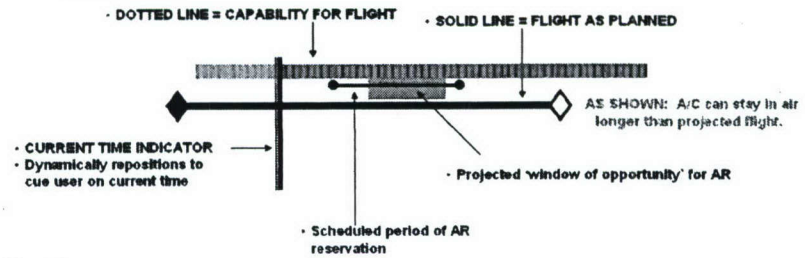
BABBA)
Name: Time Points
Link: Time-Points



Attributes:
1) Location = Along solid and dashed lines
2) Base color = white
3) Shape = circle for planned reporting time point,
square for unplanned reporting time point, diamond for geo-referenced waypoint
a) *Event:* Time point flagged
Action: Change color of time point from white to black
Subsumed Parts: 4

BABBB)
Name: Timeframes

Link: Timeframes



Attributes:
1) Display type = lines/bars
Subsumed Parts: 5

BABBBA)
Name: Flight Capability Timeframe
Link: Timeframes
Attributes:
1) Location = top of central timeline window (BABB)
2) Line type = dotted
Subsumed Parts: 0

BABBBB)
Name: Flight as Planned Timeframe
Link: Timeframes
Attributes:
1) Location = center of central timeline window
2) Line type = solid black
3) End points = diamond time points
Subsumed Parts: 0

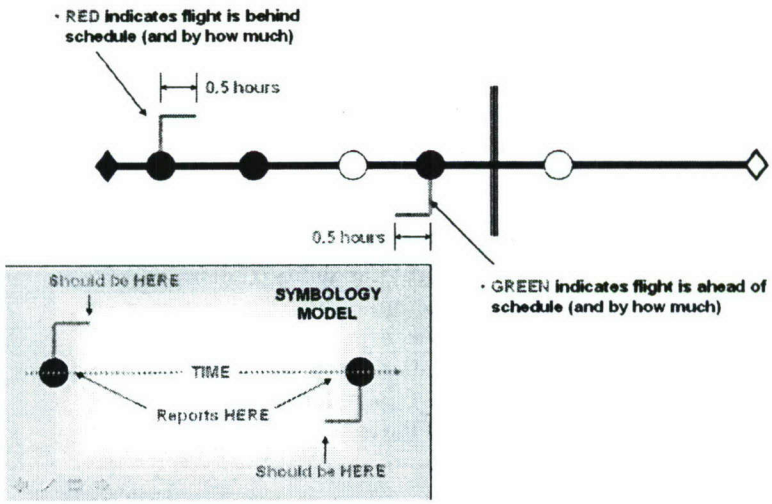
BABBBC)
Name: AR Scheduled Reservation Timeframe
Link: Timeframes
Attributes:
1) Line type = solid black
2) End points = black circles
Subsumed Parts: 0

BABBBD)
Name: AR Window of Opportunity Timeframe
Link: Timeframes
Attributes:
1) Line type = colored bar
a) *Event:* **CALL** go/no-go reservation
Action: Bar color change
Subsumed Parts: 0

BABBBE)
Name: On Ground Timeframe
Attributes:
1) Line Type = Dashed
2) Location = center of central timeline window (BABB)

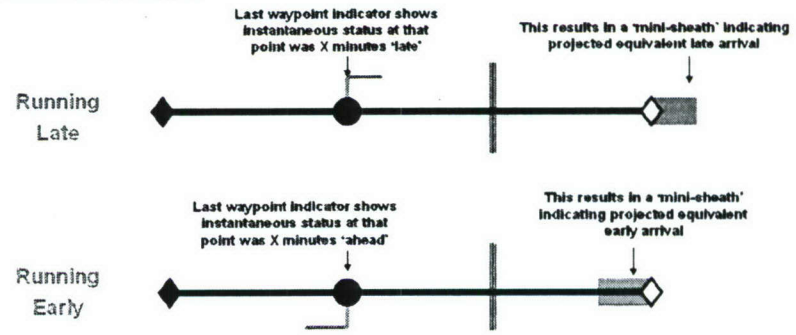
inline with Flight as Planned Timeframe (BABBBB)
Subsumed Parts: 0

BABBC)
Name: Status Indicators
Link: Status-Indicators



Attributes:
1) Location = attached to time point where time difference exists
2) Size = difference between projected time and actual time
3) Color = red for negative difference, green for positive difference
4) Direction = right for negative difference, left for positive difference
a) *Event:* **CALL** content feed
Action: Modify static attributes to accurate values
Subsumed Parts: 0

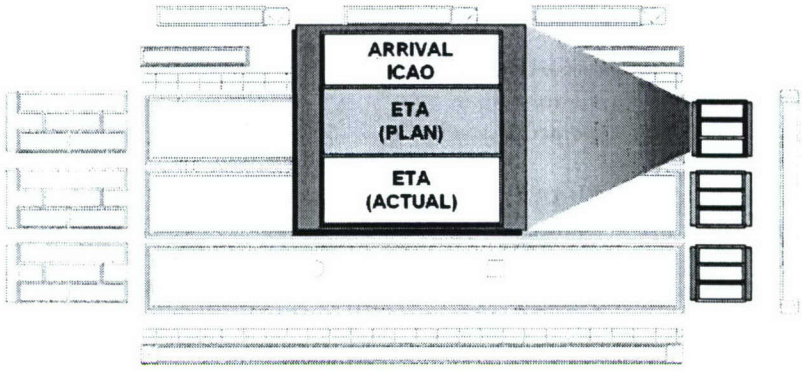
BABBD) Name: Status Bars
Link: Status-Bars



Attributes:
1) Location = attached to end of flight as planned timeframe
2) Size = difference between project time and actual time
3) Direction = right for negative difference, left for positive difference
a) *Event:* **CALL** content feed
Action: Modify static attributes to accurate values
Subsumed Parts: 0

BAC)

Name: Arrival Data
Link: Arrival-Data



Attributes: None
Subsumed Parts: 4

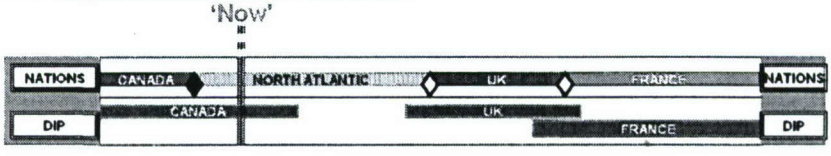
BACA)
Name: Arrival ICAO
Link: Arrival-Data
Attributes:
1) Location = Center of Arrival Data (BAC)
Subsumed Parts: 0

BACB)
Name: ETD (PLAN)
Link: Arrival-Data
Attributes:
1) Color = Shaded different than other parts within Arrival Data (BAC)
2) Location = lower left of Arrival Data (BAC)
Subsumed Parts: 0

BACC)
Name: ETD (ACTUAL)
Link: Arrival-Data
Attributes:
1) Location = lower right of Arrival Data (BAC)
Subsumed Parts: 0

BB)
Name: Cluster Display
Attributes: None
Subsumed Parts: 6

BBA)
Name: Diplomatic Permissions Cluster
Link: Diplomatic-Permissions-Cluster



Attributes: None
Subsumed Parts: 3

BBAA)
Name: DIP End Tab
Attributes: None
Subsumed Parts: 2

BBAAA)
Name: Nations Box
Attributes:
1) Location = top of DIP Cluster (BBA)
Subsumed Parts: 0

BBAAB)
Name: DIP Box
Attributes:
1) Location = bottom of DIP Cluster (BBA)
Subsumed Parts: 0

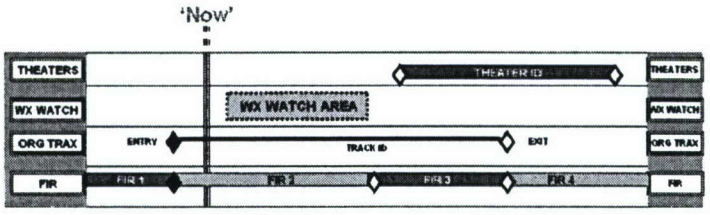
BBAB)
Name: DIP Time Window
Attributes:
a) *Event:* Disagreement of DIP/nation clearance
Action: Change status colors to reflect disagreement
Subsumed Parts: 2

BBABA)
Name: Nation Overflight Indicators
Attributes:
1) End Points = Diamond Time Points
2) Indicator Type = Horizontal Color Bar
a) *Event:* Multiple Nation Overflights
Action: Different color shading for each overflight
Subsumed Parts: 0

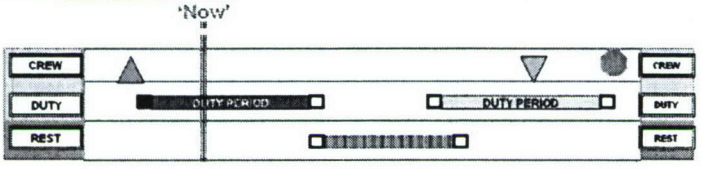
BBABB)
Name: DIP Clearance Indicators
Attributes:
1) Indicator Type = Horizontal Color Bar
a) *Event:* Multiple DIP Clearances Present
Action: Different color shading for each indicator
Subsumed Parts: 0

BBAC)
Name: DIP End Tab
Link: BBAA

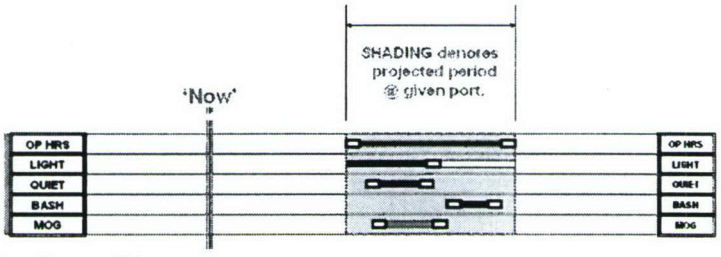
BBB)
Name: Geographical Cluster
Link: Geographical-Cluster



BBC) Name: Aircrew Cluster
Link: Aircrew-Cluster



BD) Name: Airfield Cluster
Link: Airfield-Cluster



Attributes: None
Subsumed Parts: 3

BBDA)
Name: Airfield End Tab
Link: Airfield-End-Tab

Attributes:
1) Height = 100 pixels
2) Width = 80 pixels
Subsumed Parts: 5

BBDAA)
Name: Op Hours Box
Link: Airfield-End-Tab
Subsumed Parts: 0

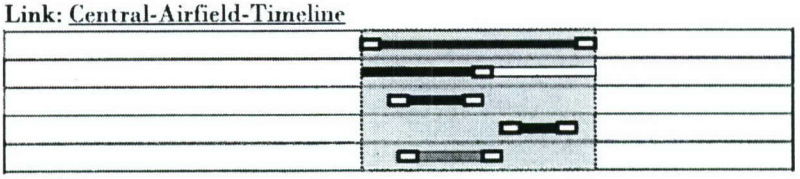
BBDAB)
Name: Light Box
Link: Airfield-End-Tab
Subsumed Parts: 0

BBDAC)
Name: Quiet Box
Link: Airfield-End-Tab
Subsumed Parts: 0

BBDAD)
Name: MOG Box
Link: Airfield-End-Tab
Subsumed Parts: 0

BBDAE)
Name: BASH Box
Link: Airfield-End-Tab
Subsumed Parts: 0

BBDB)
Name: Central Airfield Timeline
Link: Central-Airfield-Timeline



Attributes:
1) Height = 100 pixels
2) Width = 520 pixels
a) *Event:* Mouse over any subsumed part
Action: Display Time Parameters Tooltip
b) *Event:* Alert/Violation
Action: Change Color-coding
c) *Event:* Port Display Region
Action: **CALL** get projected arrival times for airfield
Subsumed Parts: Multiple

BBDBA)
Name: Individual Timelines
Link: Individual-Timeline

Attributes:
a) *Event:* Content Data Updates
Action: **CALL** respective data sources for subsumed parts
Subsumed Parts: 5

BBDBAA)
Name: OPS Timeline Bar
Link: Individual-Timeline
Subsumed Parts: 0

BBDBAB)
Name: Light Timeline Bar
Link: Individual-Timeline
Subsumed Parts: 0

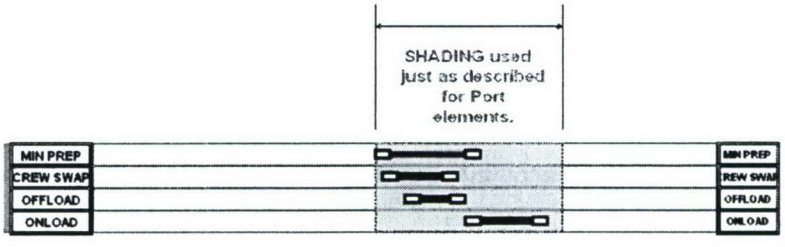
BBDBAC)
Name: Quiet Timeline Bar
Link: Individual-Timeline
Subsumed Parts: 0

BBDBAD)
Name: MOG Timeline Bar
Link: Individual-Timeline
Subsumed Parts: 0

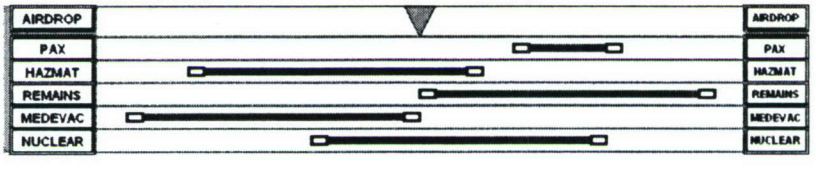
BBDBAE)
Name: BASH Timeline Bar
Link: Individual-Timeline
Subsumed Parts: 0

BBDC)
Name: Airfield End Tab
Link: BBDA

BBD)
Name: Ground Events Cluster
Link: Ground-Events-Cluster



BBE)
Name: Load/Cargo Cluster
Link: Load/Cargo-Cluster



BC)
Name: Time Indicator
Link: ABBB

REFERENCES

[1] I. Sommerville, Software Engineering, 7th edition, Edinburgh: Addison-Wesley, 2004.

[2] M. Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edition, Edinburgh: Addison-Wesley, 2004.

[3] J. Wampler, "Work-Centered Software Engineering," HE-8 Solicitation, 2004.

[4] R. G. Eggleston, "Work-Centered Support System Design: Using Frames to Reduce Work Complexity," Baltimore, 2002.

[5] C. Rourke, "Making UML the Lingua Franca of Usable System Design," Interfaces Magazine from the British HCI Group, 2002.

[6] R. G. Eggleston, M. J. Young, and R. D. Whitaker, "Work-Centered Support System Technology: A New Interface Client Technology for the Battlespace Infosphere," 2000.

[7] R. G. Eggleston, "Combining Representational and Automation Methods to Aid Complex Work," In Symposium Proceedings on Analysis, Design, and Evaluation of Human-Machine Systems HMS, International Federation of Automatic Control, Atlanta, 2004.

[8] R. Scott, E. Roth, S. Deutsch, E. Malchiodi, T. Kazmierczak, R. G. Eggleston, S. Kuper, and R. D. Whitaker, "Work-Centered Support Systems: A Human-Centered Approach to Intelligent System Design," IEEE Intelligent Systems, 2003.

[9] R. G. Eggleston, "Work-Centered Design: A Cognitive Engineering Approach to Systems Design," In Human Factors and Ergonomics Society 47th Annual Meeting, Denver, 2003.

[10] S. Lauesen, Software Requirements Styles and Techniques, Edinburgh: Addison-Wesley, 2002.

[11] J. Wampler, R. D. Whitaker, E. Roth, R. Scott, M. Stilson, and G. Thomas-Meyers, "Cognitive Work Aids for C2 Planning: Actionable Information to Support Operational Decision Making," In 2005 International Command and Control Research and Technology Symposium (ICCRTS), 2005.

[12] R. D. Whitaker, "Timeline Tool Design Concept Presentation," WIDE Project Materials, 2005.

[13] L. Burman, A. Navaro, and C. White, Mastering XML, Alameda, CA: Sybex, 2000.

[14] M. Abrams and J. Helms, User Interface Markup Language Draft Specification, Version 3.1, 2004, Available at: http://www.oasis-open.org/committees/download.php/5937/uiml-core-3.1-draft-01-20040311.pdf.